SISTEMAS INFORMÁTICOS

DYNJA: A Dynamic Resource Analyzer for Multi-Threaded Java

2012/13

Universidad Complutense de Madrid

Facultad de Informática

Alumnos:

Iván Troyano
U. Complutense of Madrid
ivantroyano@ucm.es

Oscar Troyano
U. Complutense of Madrid
oscartroyano@ucm.es

Directora:

Elvira Albert
U. Complutense of Madrid
elvira@fdi.ucm.es

ÍNDICE

LISTADO DE PALABRAS CLAVE

- JVMTI
- Multi-thread (Multi-hilo)
- Profiler (Perfilador)
- Bytecode
- Instrumentation (Instrumentación)

A través del presente documento autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales, la presente memoria, y el código del proyecto descrito en la misma, cualquier tipo de documentación y el propio desarrollo realizado por Óscar Troyano Rollán con DNI 50984244Y e Iván Troyano Rollán con DNI 50984243M.

Óscar Troyano Rollán

Fdo:

Iván Troyano Rollán

Fdo:

1. *Concepto*

Presentamos a continuación el concepto, el uso y la implementación prototípica de Dynja, un analizador dinámico de consumo de recursos para programas Java multi-hilo. El sistema recibe como entrada una aplicación Java, los valores iniciales de sus parámetros de entrada, y con ello se calculan y se miden las siguientes tres métricas disponibles actualmente: número de instrucciones ejecutadas de bytecode (código de bytes), número (y tipo) de los objetos creados, y el número (y nombre) de los métodos invocados. Dynja proporciona como salida los recursos consumidos por cada hilo de acuerdo con la métrica(s) seleccionada(s).

Nuestro analizador dinámico de recursos se ha implementado haciendo uso del framework Java Virtual Machine Tool Interface (JVMTI), un interfaz de programación nativo que permite inspeccionar el estado y controlar la ejecución de las aplicaciones que se ejecutan en una JVM.´

Las principales conclusiones del presente trabajo se han enviado para su evaluación al congreso “Principies and Practice of Programming in Java (PPPJ’13)” y actualmente se encuentra en proceso de revisión. El artículo se puede encontrar en el apéndice.

2. Concept (English)

We present the concepts, usage and prototypical implementation of Dynja, a dynamic resource analyzer for multi-threaded Java. The system receives as input a Java application, initial values for its input parameters, and the cost metrics to be measured among the three metrics currently available: number of executed bytecode instructions, number (and type) of objects created, and number (and name) of methods invoked. Dynja yields as output the resources consumed by each thread according to the selected metric(s).

Our dynamic resource analyzer has been implemented using the Java Virtual Machine Tool Interface (JVMTI), a native programming interface which allows inspecting the state and controlling the execution of applications running in a JVM.

The main conclusions of this work have been submitted for assessment to Congress "Principles and Practice of Programming in Java (PPPJ'13)" and is currently under review. The article can be found in the appendix.

3. Motivación

Durante el desarrollo de aplicaciones, tarde o temprano se suele plantear la cuestión del rendimiento de dicha aplicación. Existen múltiples métricas que pueden ser tenidas en cuenta como unidad de medida de rendimiento de una aplicación, y en función de los objetivos del desarrollo es necesario conseguir unos resultados u otros, por ejemplo:

- El número de peticiones que es capaz de responder en un periodo de tiempo.
- El tiempo mínimo necesario para responder a una petición.
- El tiempo máximo para responder una petición.
- Valores medios de un grupo de respuestas.
- Consumo de memoria

Para medir el rendimiento de una aplicación, incluyendo el número de peticiones que se responden y la memoria necesaria para responder las peticiones se suele utilizar herramientas que realizan tests de rendimiento o stress, que permiten definir un conjunto de pruebas a realizar, y muestran estadísticas de los resultados.

Una vez que se ha medido el rendimiento de la aplicación, cuando se plantea la necesidad de la mejora del rendimiento de una aplicación, es posible seguir tres vías complementarias:

- Optimización o ampliación del hardware o software contenedor.
- Crecimiento horizontal de la arquitectura de aplicación. Diseñando una arquitectura que nos permita incluir nuevos servidores para aumentar el número de peticiones que el sistema es capaz de servir.
- Y finalmente recurrir a la optimización del propio código fuente de la aplicación. En estos casos suele ser necesario recurrir o bien a procedimientos de inspección de código fuente, complementados con otros procedimientos de medición de rendimiento durante el funcionamiento de la aplicación denominados "profiling" que permiten determinar que optimizaciones serían las que aporten mayor rendimiento.

Las herramientas de análisis de rendimiento, surgieron a principios de la década de 1970, en las plataformas IBM/360 e IBM/370, basadas generalmente en interrupciones del temporizador puesto para detectar los puntos calientes en la ejecución de código.

A principios de 1974, los simuladores de conjuntos de instrucciones comenzaron a realizar una traza completa del proceso, así como funciones de supervisión del rendimiento.

Las herramientas para el análisis de programas en Unix, se remontan al menos a 1979, cuando los sistemas Unix incluyeron la herramienta básica prof, la cual lista cada función y la cantidad

de tiempo de ejecución del programa utilizado. En 1982, la función gprof extendido el concepto a un análisis completo del grafo de llamadas.

En 1994, Amitabh Srivastava y Alan Eustace of Digital Equipment Corporation publicaron un artículo en el que se describe la plataforma ATOM. ATOM es una plataforma para convertir un programa, en su propio perfilador. Es decir, en tiempo de complicación, inserta el código en el programa a ser analizado. Dicho código muestra el análisis de la información. Esta técnica (la modificación de un programa para que se analice a si mismo) se conoce como 'Instrumentation (Instrumentación). En 2004, tanto el gprof como ATOM, aparecieron en la lista de los 50 trabajos publicados en el congreso PLDI, más influyentes de todos los tiempos.

En ingeniería de software el análisis de rendimiento, comúnmente llamado profiling, es la investigación del comportamiento de un programa de computadora usando información reunida del análisis dinámico del programa (mientras éste está en funcionamiento) en oposición al análisis estático (análisis de código). La meta del análisis de rendimiento es determinar que partes del programa se pueden optimizar con el fin de obtener una mayor velocidad de procesamiento, un mejor rendimiento u optimizar el uso de memoria.

Usualmente el análisis de rendimiento tiene una gran importancia. Se deduce como buena práctica de desarrollo, que el profiling ocupe un 90% del tiempo de la realización de una aplicación. Una vez que se dispone de un prototipo o un fragmento funcional del programa podemos determinar los cuellos de botella del programa para intentar optimizarlo. El perfilado permite contestar preguntas como: ¿Dónde se gasta la mayor parte del tiempo de ejecución? De cara a concentrar los esfuerzos de optimización donde más se notara. ¿Cuál es el camino crítico? Para incrementar las prestaciones globales. Por ejemplo, el número de frames por segundo. ¿Cuál es la tasa de fallos de la memoria caché? Con el objetivo de mejorar la localidad de la memoria.

Normalmente recabar este tipo de información implica instrumentar el código añadiendo algunas instrucciones que permiten acumularla en un archivo (o varios) para cada ejecución del programa.

La información de perfilado es posteriormente analizada con un programa, el perfilador o profiler.

La salida que produce la herramienta de perfilado, es un rastro (stream) de eventos o un sumario estático de los eventos observados (un "profile", perfil o reseña). Las herramientas de perfilado usando una amplia variedad de técnicas para obtener información, incluyendo interrupciones hardware, instrumentación del código, simulaciones de conjuntos de instrucciones, contadores de rendimiento, entre otros.

El objetivo del presente proyecto consiste en la realización de un nuevo perfilador para Java multithreading. La novedad y la meta del proyecto consiste en realizar una solución independiente de la plataforma o el sistema que realiza la ejecución del proceso analizado. Para lograr dicho objetivo se realiza un análisis más abstracto, dejando de lado los detalles

técnicos que dependen de la arquitectura o tecnología del sistema, y centrándose en un análisis de cuestiones que afectan al diseño de algoritmos y a la eficacia de los métodos.

Para ello el siguiente proyecto se ha centrado en la obtención de la información concerniente a un análisis conceptual de los métodos del proceso, los cuales incluyen la obtención del número de objetos creados, su tipo, el marco en el que fueron generados, la cantidad de bytecodes de un método y los que fueron realmente ejecutados, entre otros datos. Todos estos análisis se realizan por cada uno de los hilos que ejecutan el programa de manera completamente independiente, obteniendo una idea clara y precisa de los elementos y el entorno que conforman el proceso sobre el que se realizan los análisis.

Es un hecho ampliamente reconocido que los programas multi-hilo inherentemente tienen un comportamiento complejo (véase, por ejemplo [8]). Una de las principales razones de este comportamiento complejo está relacionado con la programación de subprocesos no determinista y el orden de preferencia. Los hilos Java se seleccionan y se eligen por prioridades. Un hilo de mayor prioridad tiene preferencia sobre un hilo de menor prioridad.

Si un hilo de mayor prioridad se va a un estado de dormición o de bloqueo, entonces el subproceso de prioridad menor puede continuar. Sin embargo, tan pronto como el hilo de mayor prioridad se despierta o se desbloquea, se interrumpe el hilo de menor prioridad y se extiende hasta que el hilo de mayor prioridad finalice, se bloquee de nuevo, o hasta que sea superado por un hilo de mayor prioridad.

La especificación del lenguaje Java permite a las máquinas virtuales que ejecuten ocasionalmente un hilo de baja prioridad en lugar de una prioridad más alta, pero en la práctica esto no es habitual. Por último, nada en la especificación del lenguaje Java especifica lo que sucede con hilos con la misma prioridad. Con todo, a nivel conceptual, sólo se puede suponer que la programación no es determinista.

La Dificultad principal con la programación no determinista y de preferencia es que, en cualquier punto de la ejecución de un hilo, la memoria compartida puede ser modificada por otro hilo ejecutándose en paralelo.

Como veremos más adelante, estas modificaciones afectan la conducta del programa y, en particular, pueden cambiar su costo o el consumo de recursos. Para construir sistemas multi-hilo más seguros, previsibles y optimizados, se necesitan herramientas que ayudan a la comprensión y verificación de programas con subprocesos múltiples, tanto en su funcionamiento, como en los aspectos no funcionales. En este trabajo, nos centramos en el consumo de recursos de los programas, una de las propiedades no funcionales más importantes.

Consideramos tres métricas de costes que nos permiten medir: el número de instrucciones ejecutadas de bytecodes, el número (y tipo) de los objetos creados, y el número (y nombres) de las invocaciones a métodos. Hay dos enfoques principales para estimar el consumo de recursos: análisis estáticos y dinámicos:

- El análisis estático pretende calcular el consumo de recursos sin ejecutar el programa, por sólo inspeccionar el código del programa. Los resultados del análisis estático deben sondear cualquier ejecución del programa. Por lo tanto, este enfoque debe tener en cuenta el consumo de recursos de todos los caminos posibles de ejecución (e intercalaciones) y evitar dejar cualquier comportamiento sin marcar. El análisis devuelve el coste del peor caso de la información obtenida para todos los caminos. Debido a la gran cantidad de posibles intercalaciones que un sistema multi-hilo grande puede tener, este tipo de análisis pierde precisión rápidamente (conducir a una estimación de recursos demasiado pesimista).

- En cambio, el análisis dinámico (también conocido como perfilado o profiling) recopila la información de funcionamiento del sistema. Puede recopilar datos sobre el comportamiento preciso y minucioso de un sistema multi-hilo en ejecución, que se puede acoplar con un procesamiento posterior para resumir y razonar acerca de los resultados observados. En general, los datos recopilados son precisos, siempre y cuando la sobrecarga de la medición sobre la ejecución del sistema no influencie los resultados. El perfilado, sin embargo, se limita a la inspección de comportamiento del programa mediante la ejecución del sistema para una entrada específica. Esta limitación significa que el perfil es útil en los casos en que una muestra de comportamiento es suficiente. Esto puede ser el caso para las estimaciones sobre el consumo de recursos en caminos estadísticamente frecuentes de ejecución, pero no es muy adecuado para garantizar el comportamiento correcto en un sistema cuando sólo una ejecución en un millón puede conducir a fallo del sistema.

El enfoque de los perfiladores puede ser clasificado como activos o pasivos. El perfilado activo requiere que la aplicación que se está midiendo genere explícitamente información acerca de su ejecución. Los perfiladores pasivos se basan en la inspección explícita de control de flujo y el estado de ejecución a través de una entidad externa. El perfilado pasivo no requiere ninguna modificación del sistema de medida, pero es más difícil de implementar.

Dynja es un perfilador dinámico, es un analizador de recursos pasivo para Java multi-hilo, que tiene, entre otras, aplicaciones interesantes para (1) entender el consumo de recursos de cada hilo en presencia de intercalaciones de hilos que pueden afectar el costo, (2) para evaluar la precisión del análisis de recursos estático que infiere el consumo de recursos en el peor caso, (3) para explicar simbólicamente la razón de los “puntos calientes” en la ejecución (es decir, qué subproceso utiliza más recursos y de qué tipo). El sistema Dynja se puede descargar desde http://costa.ls.fi.upm.es/dynja donde los ejemplos utilizados se pueden encontrar también.

4. Estado del arte

La optimización de código es difícil, requiere tiempo, requiere diseño e investigación por parte de los desarrolladores. Sin las herramientas adecuadas, los programadores tienen que recurrir a formas más lentas y menos eficientes de tratar de optimizar sus aplicaciones. Normalmente los desarrolladores con el código "pre-optimizado", y antes de que se produzcan los problemas de rendimiento, prevén los posibles problemas que se pueden producir en su código, en un intento de eliminar los problemas antes de que aparezcan. Este enfoque es problemático y no es eficiente debido a que un desarrollador, a menudo incorrectamente, diagnostica los potenciales cuello de botella. Se puede dar el caso de que busque sólo en su propio código, y no en el código completo, por lo tanto no se tienen en cuenta los problemas de integración. Puede ser que no tenga información clara sobre el comportamiento esperado, o puede centrarse en un área de código que se utiliza con poca frecuencia.

Aunque bien intencionado, este enfoque general, no encuentra los cuellos de botella de rendimiento reales. Sin una herramienta de análisis para ayudar a localizar los cuellos de botella, la optimización ciega sólo una pérdida de tiempo, como se muestra en el gráfico siguiente.

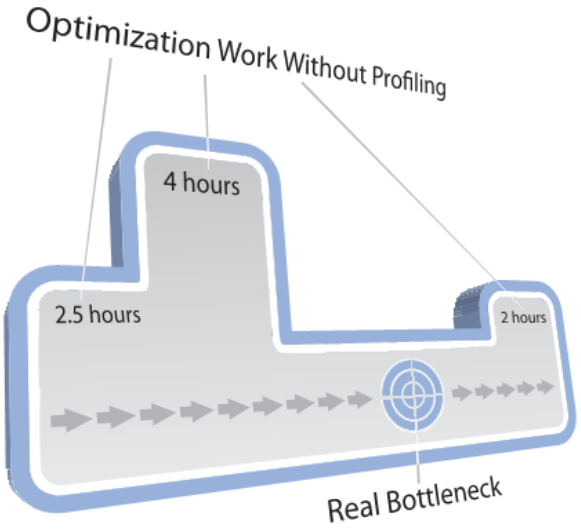


Ilustración 1 Optimización a Ciegas (http://smartbear.com/images/techpapers/how-to-choose/without-profiling.png)

Si se elige no optimizar el código es una mala idea. Ningún usuario quiere utilizar una aplicación lenta. Si estos se sienten frustrados, buscan soluciones alternativas. Los usuarios descontentos suelen expresar sus opiniones en blogs y otros foros, la dirección de los clientes potenciales de productos de una empresa a otra. Este tipo de daño a la reputación de una empresa puede ser irreparable, y puede costar a la empresa un sinnúmero de ingresos perdidos.

Es por ello que los perfiladores son actualmente un requisito, una necesidad más que un complemento o un simple analizador de costes. El continuo análisis de los costes, de los algoritmos, del tiempo, así como su rediseño y mejor, es fundamental para obtener un software final de calidad.

En la Ingeniería del Software, el análisis del rendimiento, es pieza fundamental, y se define como los exámenes que se realizan para determinar lo rápido realiza una tarea, un sistema, en condiciones particulares de trabajo. Sirve para validar y verificar otros atributos de la calidad del sistema, tales como la escalabilidad, fiabilidad y uso de los recursos. Estos análisis son una práctica informática que se esfuerza por mejorar el rendimiento, englobándose en el diseño y la arquitectura de un sistema.

Las técnicas de perfilado pueden servir para diferentes propósitos. Pueden demostrar que el sistema cumple los criterios de rendimiento. Pueden medir que partes del sistema o donde se encuentras los lugares de mayor carga de trabajo. Es fundamental para alcanzar un buen nivel de rendimiento de un nuevo sistema, que los esfuerzos en estos análisis sean importantes, persiguiendo continuamente la mejora y optimización.

En el perfilado, a menudo es crucial (y con frecuencia difícil de conseguir) que las condiciones de análisis sean similares a las reales y acaparen todos los posibles contextos y vías de análisis posibles. Esto conlleva, la mejora de rendimiento de todo el sistema, detectando sus posibles errores y mayores cargas de trabajo, pudiendo mejorar tanto el espacio en memoria como la velocidad de procesamiento.

En el mercado podemos encontrar distintas herramientas de perfilado:

- **Visual VM**

VisualVM es una herramienta visual que integra varias herramientas JDK de línea de comandos y capacidades de perfilado ligeros. Diseñado para el análisis y estudio del uso de las aplicaciones, ofreciendo la capacidad de monitoreo y análisis de rendimiento para la plataforma Java SE.

VisualVM está diseñada para:

- Application Developer: Monitoreo, perfilado, vertederos, volcado de almacenamiento dinámico.
- Administrar el sistema: aplicaciones monitor Java y control de red.
- Java Application User: Creación de informes de errores que contienen toda la información necesaria

Características:

- Muestra las aplicaciones locales y remotas de Java. VisualVM automáticamente detecta y enumera forma local como remota la ejecución de aplicaciones Java. También puede definir las aplicaciones manualmente mediante conexión JMX. De esta manera se puede ver fácilmente lo que las aplicaciones Java ejecutan en el sistema o comprobar si un proceso de servidor J2EE remoto está activo.

- Muestra la configuración de la aplicación y el entorno en tiempo de ejecución. Para cada aplicación suministrada a VisualVM, muestra información en tiempo de ejecución básica: PID, la clase principal, los argumentos pasados al proceso java, versión JDK, las "flags" suministradas a JVM y los argumentos y las propiedades del sistema.

- Se ejecuta en Oracle / Sun JDK 6, pero es capaz de monitorear aplicaciones que se ejecutan en JDK 1.4 y superior. Se utilizan diversas tecnologías disponibles, como jvmstat, JMX, el Agente de mantenimiento (SA), y el API de Acople para obtener los datos de forma automática y utiliza estas soluciones para imponer una sobrecarga mínima en aplicaciones supervisadas.

- Tomar y ver volcados de almacenamiento dinámico. Cuando tenga que examinar el contenido de la memoria de la aplicación o descubrir una pérdida de memoria en su aplicación, se puede utilizar la herramienta HeapWalker incorporada. Puede leer los archivos escritos en formato y también es capaz de navegar por vuelcos de almacenamiento dinámico creado por la JVM en una OutOfMemoryException.

- Analizar los volcados de memoria. Cuando se bloquea un proceso de Java, un volcado de memoria puede ser generado por el JVM que contiene información importante sobre el estado de la aplicación en el momento del accidente. VisualVM es capaz de

mostrar configuración de la aplicación y el entorno de ejecución, así como la memoria almacenada.

- Analizar las aplicaciones offline. VisualVM es capaz de guardar la configuración de la aplicación y el tiempo de ejecución, los volcados de almacenamiento dinámico y snapshots del perfilado en una sola instantánea de la aplicación, que luego pueden ser procesados sin conexión. Esto es especialmente útil para los informes de error que los usuarios pueden proporcionar un único archivo que contiene toda la información necesaria para identificar el entorno de ejecución y el estado de la aplicación.

Con todo ello, persigue encajar perfectamente en todos los requisitos de los desarrolladores de aplicaciones, administradores de sistemas, ingenieros de calidad y - por último, pero no menos importante – los usuarios de aplicaciones que pueden visualizar informes de errores que contienen toda la información necesaria.

- **JProfiler**

JProfiler es una herramienta de perfilado Java con licencia comercial de desarrollo por EJ-technologies GmbH, dirigida a Java EE y Java SE.

JProfiler funciona tanto como una aplicación independiente y como un plug-in para el entorno de desarrollo de software Eclipse.

JProfiler soporta perfiles locales (análisis de las aplicaciones que se ejecutan en la misma máquina que el software JProfiler) y el perfil remoto (análisis de las aplicaciones Java que se ejecutan en equipos remotos).

Permite el perfilado de memoria para evaluar el uso de memoria y las fugas de asignación dinámica y perfiles de CPU para evaluar los conflictos de rosca.

Proporciona una representación visual de la carga de la máquina virtual en términos de bytes activos y totales, instancias, hilos, clases y actividades recolector de basura.

- Vivir perfiles de sesión local

Una vez definida la forma en que se inicia la aplicación, JProfiler perfila y se pueden visualizar inmediatamente los datos en tiempo real de la JVM perfilada. Para eliminar la necesidad de configuración de sesión, puede utilizar uno de los muchos plugins IDE para perfilar la aplicación desde dentro de su IDE favorito.

- Vivir perfiles de una sesión remota

Mediante la modificación de los parámetros de la VM de la orden de inicio de Java se puede conseguir que cualquier aplicación Java pueda realizar una conexión desde el JProfiler GUI. La

aplicación perfilada no sólo se puede ejecutar en su equipo local, JProfiler puede adjuntar a la solicitud de perfilado en la red. Además, JProfiler ofrece numerosos asistentes de integración para todos los servidores de aplicaciones populares que le ayudan en la creación de su aplicación para la creación de perfiles.

- Desconectado

No hay que conectarse con el interfaz gráfico de JProfiler, de la aplicación de perfilado para perfilar aplicaciones particulares: Con dicha opción se puede utilizar el sistema de acceso rápido de JProfiler o la API JProfiler para controlar el agente de perfiles y guardar instantáneas en el disco. En un momento posterior se puede abrir estas instantáneas en el JProfiler GUI o mediante programación exportar vistas de perfiles con la herramienta de exportación de línea de comandos.

- Comparación instantáneas

En JProfiler, se puede guardar una instantánea de todos los datos del perfilado actual en el disco. JProfiler dispone de comparación ricos para ver lo que ha cambiado entre dos o más instantáneas. Alternativamente, usted puede crear informes de comparación mediante programación con la herramienta de comparación de línea de comandos.

- Visualización de instantáneas

JProfiler puede abrir instantáneas que se han tomado con herramientas de JVM como jconsole o jmap.

- Solicitud de seguimiento

Con el concepto de seguimiento de solicitudes, JProfiler conecta las llamadas con los métodos de ejecución entre los diferentes temas con hipervínculos en la vista de árbol de llamadas. Los siguientes sistemas multiproceso pueden ser seguidos:

  - Ejecutores del paquete java.util.concurrent
  - Eventos AWT
  - Eventos SWT
  - Comienza Tema

- Fácil creación de sondas personalizadas

JProfiler ofrece un asistente de sondeo personalizado que permite definir sus sondas personalizadas directamente en la JProfiler GUI. Sus sondas personalizadas son desplegadas en la aplicación perfilada por JProfiler y ni siquiera es necesario que reiniciar la aplicación perfilada al cambiar o añadir sondas personalizadas.

- Perfiles de CPU

JProfiler ofrece varias maneras de grabar el árbol de llamadas para optimizar el rendimiento o detalle. El hilo o grupo de hilos, así como el estado de rosca pueden ser elegidos para todas las vistas. Todos los puntos de vista pueden ser agregados en un método, clase, paquete o nivel de componentes Java EE. La sección vista CPU contiene:

- Llame árbol

Muestra un árbol de arriba hacia abajo acumulado de todas las secuencias de llamadas registradas en la JVM. Llamadas de servicio JDBC, JMS y JNDI se anotan en el árbol de llamadas. El árbol de llamadas se puede dividir para diferentes URL de petición a un servlet o JSP. Puede marcar los métodos de "excepcional grabación método run" y ver las invocaciones más lentas por separado. Con la solicitud de seguimiento, se puede conectar sitios de llamada a los sitios de ejecución de las aplicaciones de subprocesos múltiples.

- Puntos calientes

Muestra la lista de los métodos que son más utilizados. El árbol de trazas se muestra para cada zona activa.

- Gráficos

Muestra un gráfico de secuencias de llamada a partir de los métodos seleccionados, clases, paquetes o componentes de Java EE.

- Métodos estadísticos

Muestra información estadística acerca de la distribución de tiempos de llamada para todos los métodos, junto con un gráfico de distribución de tiempo de llamada, que se puede utilizar para detectar valores atípicos.

ej-TECHNOLOGIES' JProfiler

Con todo ello, esta herramienta es uno de los perfiladores de referencia en el mercado.

- **JBoss Profiler**

JBoss Profiler es un perfilador de registro basado en el uso JVM (Java Virtual Machine) y JVMTI (Java Virtual Machine Tool Interface) y orientado a páginas Web. Utiliza un agente escrito en C que captura los eventos de la JVM y los registros en el disco. Una aplicación web que se ejecuta en JBoss u otro tipo de máquina se puede utilizar para analizar estos registros a través de un navegador web.

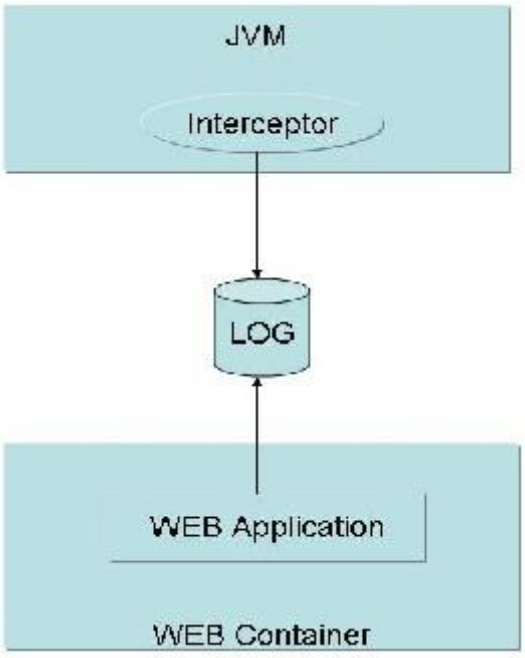
JBoss Profiler hace uso de los archivos de registro y es especialmente útil para el análisis de un servidor de aplicaciones. Creación de instantáneas de perfilado sin la necesidad de un front-end cerca de la JVM significa que los datos pueden ser analizados remotamente.

En el caso que el servidor de aplicaciones se ralentice, o tenga problemas. JBoss Profiler puede ejecutarse y analizar los datos del navegador web. Sin tener necesidad de instalar un entorno de herramientas complejas o tener que enviar datos a través de un puerto abierto, rompiendo las reglas del cortafuegos entre el generador de perfiles de front-end y el JVMPI / JVMTI.

Su funcionamiento es simple, un interceptor envía eventos a un archivo LOG. Otra aplicación (aplicación WEB) lee su contenido y analiza los datos.

La mayor parte del trabajo es realizado por la herramienta de análisis, lo que significa que el trabajo de análisis de la solicitud se hace fuera de tiempo de ejecución de la JVM. Lo que es una gran ventaja para los usuarios, pues esto aumenta la posibilidad de análisis y el usuario termina capturando más información.

Además, la idea de que el interceptor es permanecer "dormido" hasta que alguien le despierte



- **Java Interactive Profiler**

Java Interactive Profiler (JIP) es un perfilador de alto rendimiento que está escrito enteramente en Java.

JIP le da a los desarrolladores la capacidad de activar y desactivar el perfilador mientras que la máquina virtual se está ejecutando. También puedes alterar las clases y los paquetes así como controlar la salida.

Estas son algunas de las características claves de "Java Interactive Profiler":

- Interactividad. Se inicia a la vez con el programa y termina cuando JVM se cierra. En muchos casos esto no te da una verdadera medida del rendimiento ya que el compilador Just In Time no compila el código desde el comienzo. Además, este tipo de perfilador no se utiliza para nada en las aplicaciones web, ya que terminas por perfilar tanto el contenedor web así como la aplicación web.Por otro lado, JIP te permite activar y desactivar el perfilador mientras que JVM se ejecuta.
- Sin código nativo. La mayoría de los perfiladores tienen algún componente nativo. Esto es porque la mayoría de los perfiladores utilizan el JVMPI (Java Virtual Machine Interface Profiling) que requiere el uso de componentes nativos. JIP, sin embargo, es puro Java.
- Se aprovecha de la función de Java5 que te permite conectar el cargador de clases. JIP agrega aspectos a cada método de cada clase que deseas perfilar. Estos aspectos le permiten capturar los datos de rendimiento.
- Sobrecarga muy baja. La mayoría de los perfiladores son muy lentos. En muchos casos, prof haría que un programa se ejecute 20 veces más lento. JIP es ligero. Una máquina

virtual con los perfiles activados es casi dos veces más lenta que una máquina sin perfilador. Cuando el perfilador se apaga, no hay casi ninguna sobrecarga asociada con el uso de JIP.

- Intervalos de rendimiento. JIP recopila los datos de rendimiento. La mayoría de los perfiladores no se pueden utilizar para hacer los horarios de la aplicación. hprof, por ejemplo, te mostraría la cantidad relativa de tiempo que se gasta en diferentes partes del código, pero hprof tiene tanta sobrecarga, que no se puede utilizar para obtener mediciones de tiempos del mundo real. JIP, por otro lado, está actualmente siguiendo la cantidad de tiempo que se utiliza para recopilar los datos de rendimiento y los factores interrumpen el análisis. Esto te permite obtener unos horarios muy cercanos al mundo real para cada clase en el código.

- Filtros por nombre de paquete/clase. Una de las cosas fastidiosas de hprof es que no hay manera de alterar las clases por el nombre del paquete o por clase. JIP te permite hacer precisamente eso. El tiempo de ejecución está incluido, pero solo puede verse en el tiempo de ejecución de la rutina.

➢ **Profiler4j**

Profiler4j es una herramienta de perfilado para Java, basada en el perfilado remoto y la configuración de opciones mientras el programa se encuentra en ejecución. La meta fundamental de Profiler4j es ser simple, utilizando y permitiendo interacciones sofisticadas de UI.

Aquí están algunas características dominantes de Profiler4j:

- Basado en la instrumentación dinámica del bytecode.

- Se pueden añadir reglas para seleccionar los que paquetes, clases, y métodos, que deben ser perfilados. Incluso, se pueden modificar dichas reglas sin tener que reiniciar la aplicación.

- Java 100%. no requiere ninguna biblioteca nativa o ejecutable.

- Despliegue simple: apenas agregar un parámetro a la línea de comando.

- Está dividido en dos partes principales:
  - Un agente de perfilado (profiling agent) que se ejecuta en la misma JVM que la aplicación.
  - Una consola remota que conecta con el agente remoto y permite al usuario visualizar los datos de perfilado y cambiar sus opciones.

- Posee un interfaz de usuario amigable. En la que se puede visualizar y analizar los siguientes datos del perfilado:
  - Un grafo de llamadas que muestra el hot path.
  - El árbol de llamadas
  - Un monitor de memoria.
  - Un monitor del estado de los hilos.
  - Una lista de las clases cargadas que pueden ser perfiladas y el estado actual de su instrumentación, y
  - Las opciones de cargar y guardar las opciones.
- Configuración de grano fino que puede ser modificada libremente sin el reinicio del JVM de la herramienta.

Esta aplicación se encuentra en fase de desarrollo, no habiendo producido todavía un producto final.

- **EUREKAJ**

EurekaJ es una herramienta, Open Source, para el perfilado de aplicaciones Java. Este proyecto se encuentra en fase beta. Habiendo lanzado varios prototipos funcionales de la aplicación.

El proyecto tiene como objetivo desarrollar un completo perfilador para aplicaciones Java, que consiste en la creación de un agente que puede ser instalado y comenzado junto con la aplicación que se quiere monitorear. Además, una aplicación de gestión, se encarga de la recepción de la información desde múltiples agentes, acumulando dicha información y haciendo posible para el desarrollador, los operadores, las aplicaciones de gestión, los técnicos, etc. conectarse y visualizar los datos en tiempo real, así como visualizar la información histórica de los últimos 30 días.

El objetivo es desarrollar una aplicación general para el monitoreo de aplicaciones Java con las siguientes metas:

- La creación de un agente funcional con la posibilidad de decidir el grado de seguimiento por aplicación (instrumentación personalizada)
- Una aplicación de gestión que sea escalable tanto en el número de agentes, usuarios y almacenamiento de datos, teniendo en cuenta las necesidades personales del usuario.
- Una completa aplicación de administración que ofrezca al usuario la posibilidad de ver la información crucial sobre el rendimiento de las aplicaciones y el consumo de recursos (memora, CPU, Threads, IO, etc), así como los posibles errores y excepciones.
- La posibilidad de configurar las alertas, para la realización de cualquier tipo de medición,
- así como la gestión del envío de alertas a través de varios canales (correo electrónico, SNMP, etc).

Los agentes envían datos a través de TCP/IP con intervalos cortos a través de HTTP / HTTPS.

El principal objetivo de EurekaJ Profilers es trabajar para realizar un perfilador Java con las siguientes características:

- De grano fino: Los métodos de bajo nivel pueden ser monitoreados.
- Consolidado: Todas las estadísticas conseguidas son dirigidas al mismo servidor lógico para tener la posibilidad de ofrecer una visión consolidada.
- Efectivo: La recopilación de datos debe tener el mínimo impacto negativo sobre el rendimiento de la aplicación supervisada.
- En tiempo real: Los datos recogidos serían visualizados, reportados y alertados en tiempo real.
- Histórico: Los datos se almacenan en 30 días para la visualización, comparación y presentación de informes de datos históricos.

- **JProbe**

JProbe, un completo conjunto de herramientas de perfil de Java, completamente rediseñado y simplificado. Combina una fácil instalación, configuración y flujo de trabajo con el más profundo análisis y más amplio soporte de la versión de Java disponible, lo que lo convierte en la solución de perfil de Java más potente del mercado.

Diagnostica y resuelve los problemas de codificación, lo que le permite ofrecer aplicaciones de alta calidad, listas para la producción, además posee las siguientes funcionalidades:

- Análisis de la memoria: muestra los problemas de distribución de memoria, incluidos el consumo de memoria, las fugas de memoria y la excesiva recolección de basura. Para que el desarrollador pueda reducir o eliminar los fallos de servidores y el estrés del recolector de basura.

- Uso de la memoria: determina el uso de la memoria mediante un potente modelado de objeto referencia con base en situaciones hipotéticas. Modela rápida y fácilmente lo que ocurriría si se liberase una referencia por la ruta de referencia. Determina sin esfuerzo el tamaño potencial de una fuga de memoria.

- Procesamiento de texto y volcados de memoria portátil: captura el texto de producción completo y las métricas de volcado de memoria portátil, sin costos operativos. Compara la memoria con otras tomas de instantáneas para identificar y aislar rápidamente las fugas de memoria y otros problemas de distribución de memoria en producción.

- Análisis de rendimiento: aísla los cuellos de botella en los códigos, monitorea los hilos de ejecución activos, descubra los bloqueos mutuos y localizarlos con precisión. Identifica las secciones más frecuentemente ejecutadas de su código, así como también aquellas que representan el mayor tiempo de ejecución.

- Análisis de cobertura: reduce el tiempo de control de calidad al mejorar el código. Verifica la terminación y exactitud de los códigos antes de que hacer funcionar la prueba JUnit.

- Promueve la cooperación entre los equipos de desarrollo y de control de calidad para crear

- paquetes de prueba más completos. Identifica fácilmente los métodos más que probados.

Estas herramientas no cumplían con las necesidades que nosotros planteábamos de análisis abstracto de la aplicación. Obteniendo los resultados conceptuales necesarios para obtener una idea clara de cómo está funcionando nuestros proceso. Todo ello, además para varios hilos ejecutándose concurrentemente. Es por ello que nos vimos y se planteó la necesidad de realizar nuestra herramienta.

5. Descripción

El sistema Dynja se puede utilizar a través de su interfaz Java que se representa en la Figura 2. El usuario selecciona primero la aplicación a analizar. Después de pulsar el botón de compilación, Dynja compila utilizando la instalación en la máquina JVM, e identifica los métodos definidos en la aplicación.

La lista de los métodos que están disponibles se muestran en la lista “Method to profile" que aparece a continuación.

Ejemplo 1. Como ejemplo, considere la siguiente clase MyThread y ResThreads. Los objetos de tipo MyThread tienen un campo ini y un campo de t que se inicia en la construcción.

Como MyThread extiende la clase Thread, los objetos de tipo MyThread se pueden ejecutar a través de método “run” que a su vez invoca el método·”task” en el campo del objeto t. Al ejecutar “task” con varios hilos las Intercalaciones concurrentes entre estos pueden ocurrir. Los elementos de interés en el método “task” son (1) el uso de datos compartidos, es decir, “bound” y “z” son campos de clase que tendrán valores diferentes dependiendo de la política de planificación y a la intercalación de los hilos, y esto afectan la operación de consumo de recursos, (2) una división por 0 en la línea 30 se puede producir en función de la programación y la intercalación de los hilos, afectando aún más el consumo de recursos.

El método “main” recibe tres datos enteros a través de sus parámetros argv entrada, crea un solo objeto a testear utilizado más tarde por los dos hilos creados en las líneas 38 y 42. Estos hilos se inician después de la creación y el método “run” comienza a ejecutar cada uno de ellos. Como los hilos comparten los datos del objeto a testear, la estrategia de programación y las intercalaciones pueden cambiar los valores de los campos del probador de manera diferente. Este afecta al consumo de recursos como se ilustra a continuación.

Podemos ver en la interfaz Java de Dynja que, si el método a analizar es el principal, los valores de entrada se proporcionan en un TextField especificando por separado los valores con espacios en blanco. En el ejemplo, hemos proporcionado como valores iniciales 1 3 2. Además, podríamos especificar opcionalmente los valores de entrada fijos para los parámetros y variables de otro método en la aplicación (que es diferente del principal). Pulsando sobre el botón “Show variables”, visualizamos todas las variables definidas en el método seleccionado, incluyendo los parámetros de entrada. Por ejemplo, se puede forzar “task” a que se ejecute siempre con unos valores iniciales de X como parámetro de entrada. Se proporciona el valor de X utilizando la sintaxis “ini = X "en el TextField correspondiente. En la Figura 2, sólo hemos proporcionado los valores iniciales de principal, así el análisis parte de la ejecución de la tarea principal y se ejecuta con los valores que los parámetros tienen realmente en las llamadas a “task” partiendo de la función “main”.

La siguiente opción se refiere a los indicadores de costos, es decir, al recurso que se requiere medir. Podemos seleccionar entre los siguientes parámetros de costos simbólicos que se miden en cada uno de los subprocesos creados (los indicadores son simbólicos en el sentido de que son independientes del entorno de ejecución):

- Bytecode: el número de instrucciones ejecutadas de bytecodes en cada hilo.
- Objects: el número de objetos creados de cada tipo;
- Calls: el número de llamadas a métodos desde el método seleccionado.
- All: todas las mediciones anteriores.

El último parámetro de entrada de Dynja permite seleccionar si la salida del análisis de tiempo de ejecución se muestra en modo “verbose” (detallado) o “compact” (compacto). El modo compacto para nuestro ejemplo se puede ver en la Figura 3, donde vemos el resultado de las métricas de coste de cada hilo, así como el tiempo de ejecución del hilo. Vamos a explicar los resultados del análisis. En primer lugar, observamos que hemos añadido un bucle en la línea 41 para permitir que el thread1, hilo de baja prioridad, comience su ejecución, pero se puede visualizar que el hilo de alta prioridad thread2, interrumpe su ejecución. Además, el bucle de la línea 23 se ejecuta sólo en thread1 (de ahí que este hilo ejecute más instrucciones bytecode que thread2). La motivación de la adición de este bucle es que nos permite capturar el hecho de que el bucle de “bound”, “bound” puede ser modificado por thread2 en el medio de la ejecución del bucle. En la salida, que de hecho se observa, debido a la ejecución del thread2, thread1 produce una excepción porque thread2 disminuye el valor de “bound” a 2 y, por lo tanto “ini” aumenta sólo hasta 3 en thread1, y entonces “r” toma el valor 0. Esto provoca una división por 0 en t1 y por lo tanto el código siguiente (que se comenta en la línea 31) no añade aún más recursos al consumo del hilo.

```
1  public class MyThread extends Thread{
2   int ini;
3   ResThreads t;
4   public MyThread(String name, ResThreads t, int ini) {
5     super(name);
6     this.t = t;
7     this.ini=ini;
8   }
9   public void run() {
10    t.task(ini);
11  }
12 }
13 public class ResThreads{
14 int z;
15 int bound;
16 public ResThreads(int z,int bound){
17    this.z=z;
18    this.bound=bound;
19 }
20 public void task(int ini){
21    while(ini<bound)
22    {
23    if (ini ==1) for(int j =0; j<5;j++);
24      Integer obj = new Integer(ini);
25      obj.intValue();
26      ini++;
27     }
28     bound--;
29    int r= z-ini;
30    int exc = 2/r;
31    // not executed if exception
32 }
33 public static void main(String argv[]){
34    int ini   = Integer.parseInt(argv[0]); //1 3 2
35    int bound = Integer.parseInt(argv[1]);
36    int z     = Integer.parseInt(argv[2]);
37      ResThreads tester = new ResThreads(z,bound);
38      Thread t1 = new MyThread("thread1",tester,ini);
39      t1.setPriority(Thread.MIN_PRIORITY);
40      t1.start();
41      for (int i=0;i<4;i++);
42      Thread t2 = new MyThread("thread2",tester,ini+1);
43      t2.setPriority(Thread.MAX_PRIORITY);
44      t2.start();
45  }
46 }
```

Figura 1 Ejemplo java de aplicación Multi-Thread

Hay que tener también en cuenta que el número de iteraciones del bucle while en la línea 21 varía en función de la intercalación de los hilos, y esto afecta el número de objetos creados en la línea 24 y el número de llamadas en la línea 25. En particular, thread1 crea sólo 1 objeto y hace 1 llamada al método intValue (el mismo while sin la modificación de thread2 debería haber consumido 2 de cada tipo), ya que "bound" se redujo en el thread2 que se ha intercalado. El modo detallado, además, permite visualizar la información del "heap" del proceso que se recoge a través del interfaz JVMTI. Un aspecto interesante es que las ejecuciones consecutivas de Dynja con los mismos datos de entrada pueden conducir a valores diferentes de consumo de recursos. Esto ocurre por ejemplo si eliminamos las dos líneas 39 y 43 con las invocaciones SetPriority y también el retraso añadido en la línea 41.

A medida que los hilos tienen la misma prioridad, algunas ejecuciones pueden comenzar por thread1 y otras por thread2, al ser no determinista.

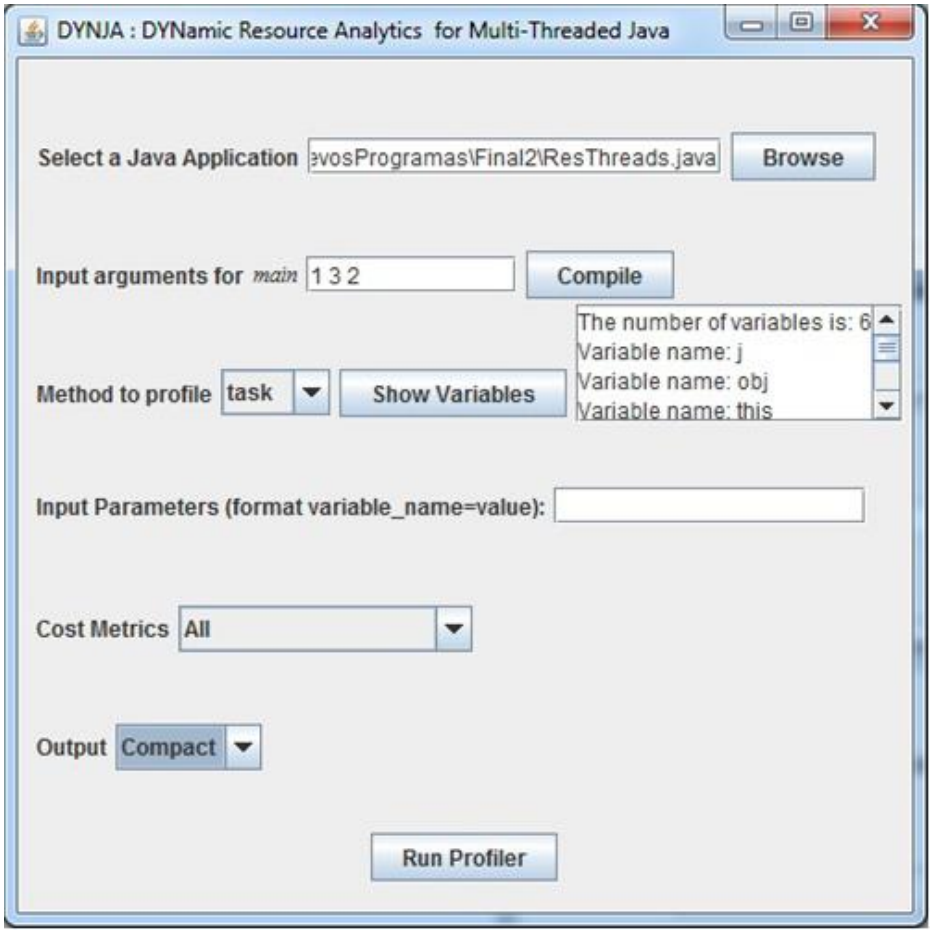


Figura 2 Interfaz Dynja

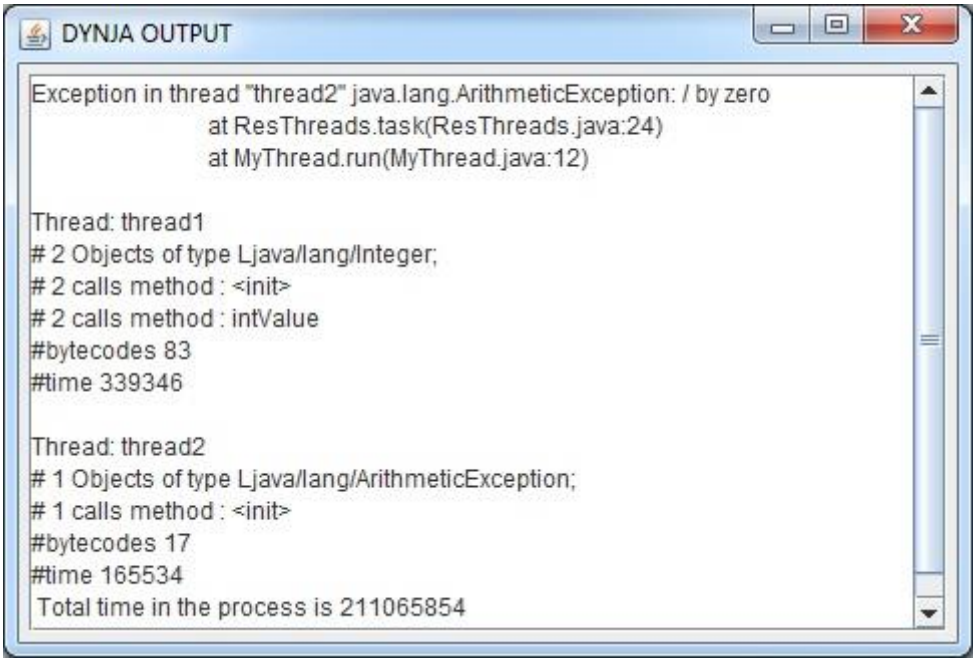


Figura 3 Datos compactos de salida de Dynja Salida 1

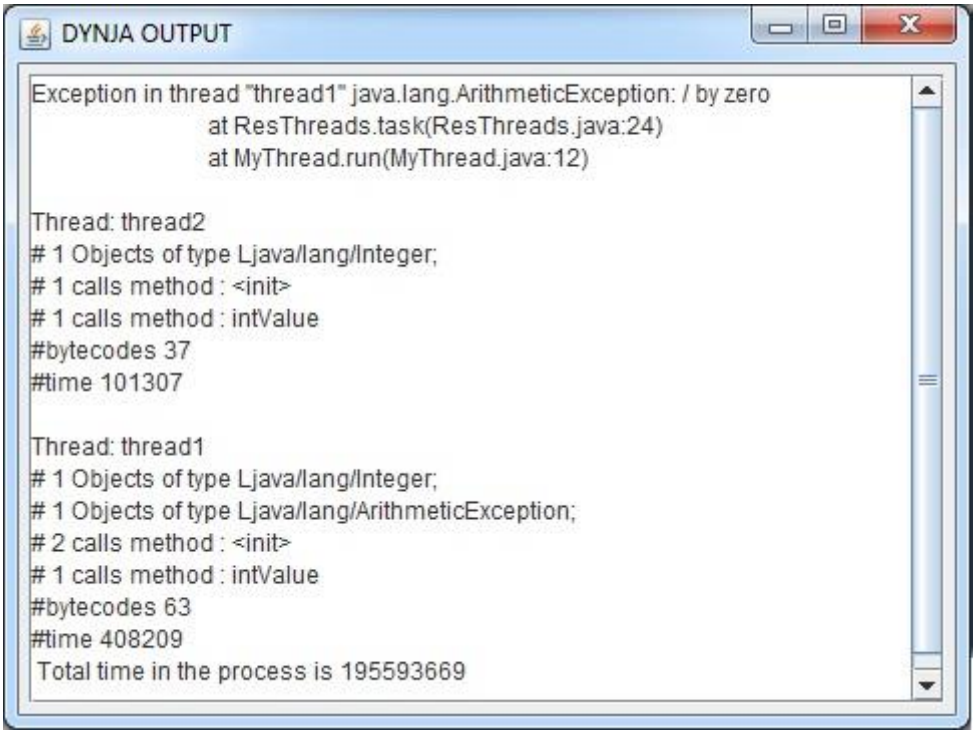


Figura 4 Datos compactos de salida de Dynja Salida 2

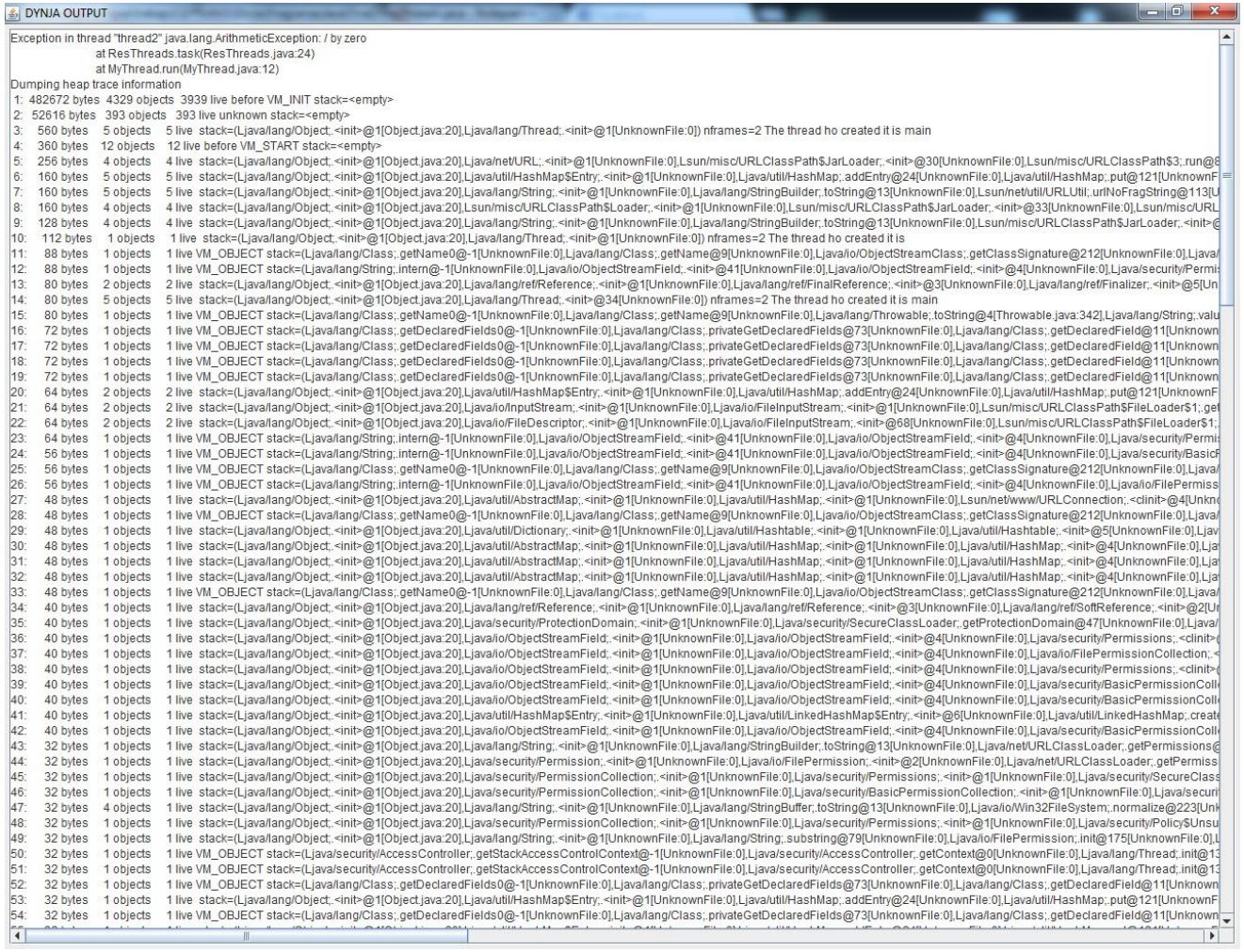


Figura 5 Datos extendidos de salida de Dynja Salida 1 (1 de 4)

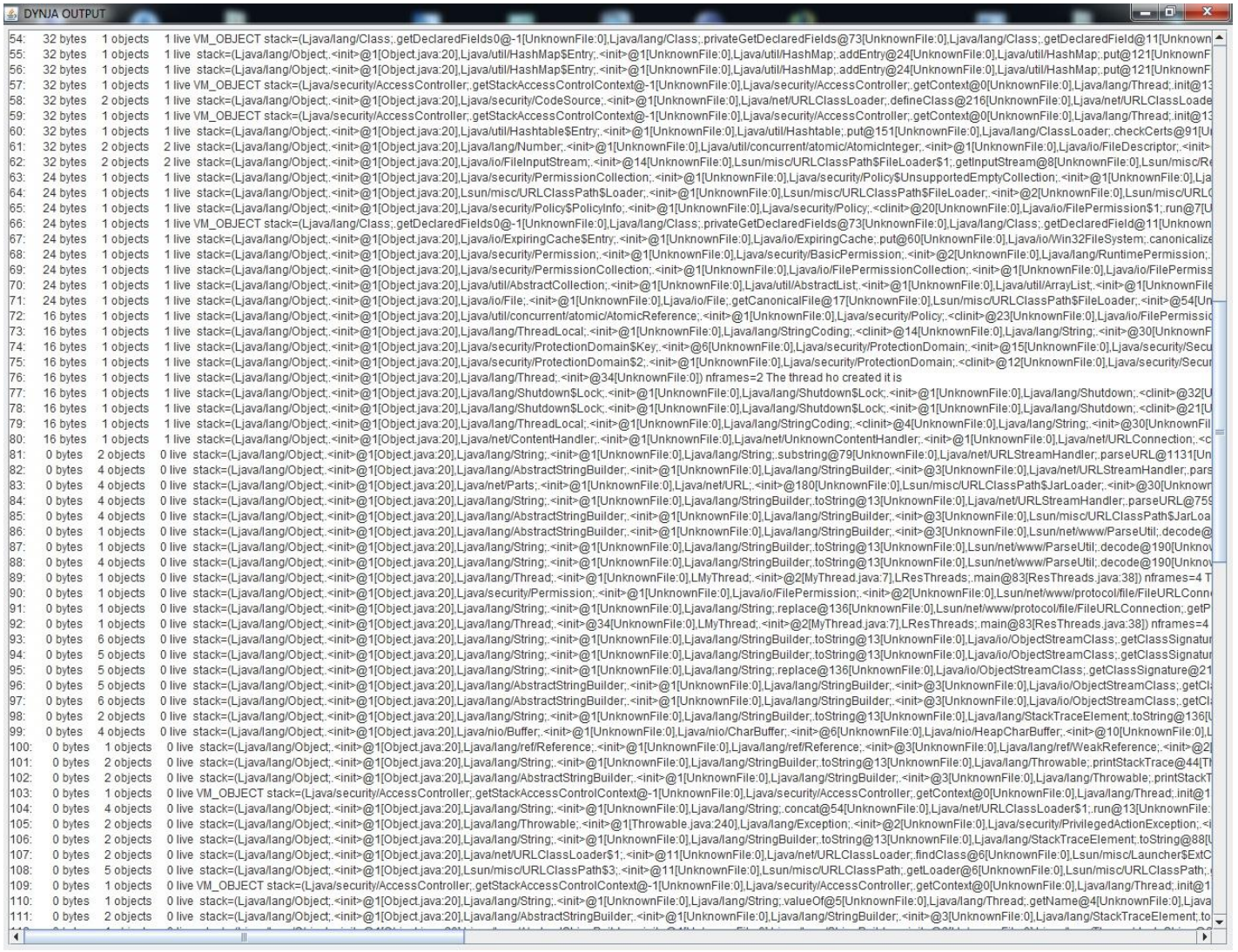


Figura 6 Datos extendidos de salida de Dynja Salida 1 (2 de 4)

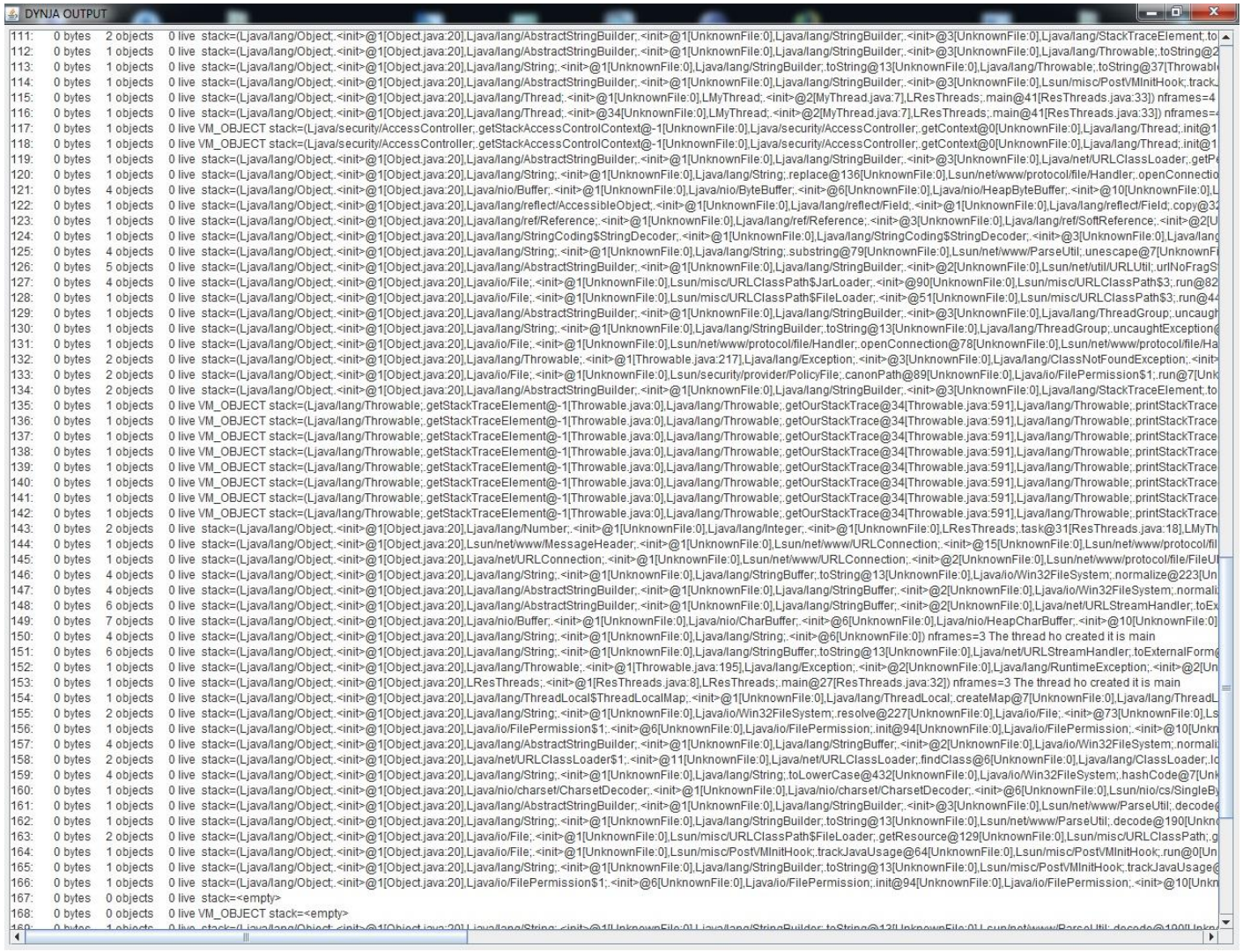


Figura 7 Datos extendidos de salida de Dynja Salida 1 (3 de 4)

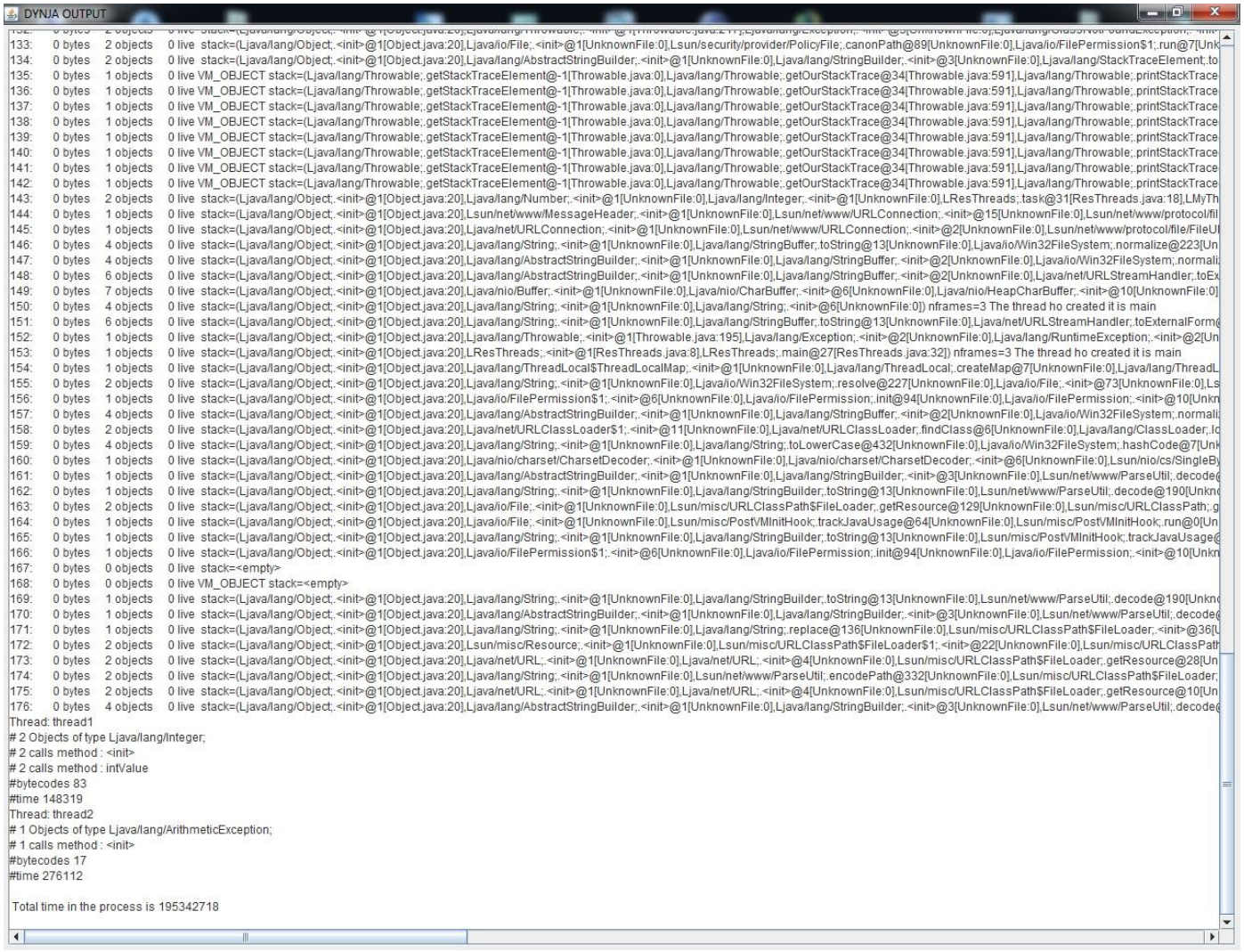


Figura 8 Datos extendidos de salida de Dynja Salida 1 (4 de 4)

6. Desarrollo

Hemos implementado la herramienta Dynja mediante la interfaz Java Virtual Machine Tool Interface (JVMTI), que proporciona acceso y control sobre la JVM en las que se ejecuta el programa.

Esto permite que el analizador esté desacoplado de la JVM que ejecuta y se puede utilizar en diferentes JVM y plataformas. JVMTI proporciona una interfaz de programación que permite la creación de agentes software, que permiten el control y el seguimiento de las aplicaciones Java. El propósito de JVMTI es permitir la creación de elementos de depuración y herramientas de perfilado para aplicaciones Java.

En cuanto a la ejecución, JVMTI es una interfaz nativa de la JVM. Una biblioteca, escrita en C o C + +, se carga durante la inicialización de la JVM. La biblioteca tiene acceso al estado de JVM llamando a funciones JVMTI y JNI (Java Native Interface) y puede registrarse para recibir eventos desde JVM, a través de funciones de control de eventos que son llamados cuando son producidos en la aplicación analizada. A pesar de los eventos predefinidos que JVMTI ya prevé, el desarrollo de Dynja ha requerido la instrumentación de las instrucciones de bytecodes, asociados a la creación de objetos o arrays con el fin de ser capaz de realizar un seguimiento de los objetos y arrays creados para cada hilo. Entonces, la parte principal de nuestra aplicación es un agente que utiliza parte de la funcionalidad expuesta por la interfaz para registrar las notificaciones de eventos asociados a nuestras métricas de coste, cuando se producen en la aplicación. Básicamente, cuando se produce un evento (por ejemplo, se crea un objeto), se agrega la información en el subproceso que lo ha creado. Cuando se ejecuta una instrucción de bytecode, se comprueba el método que se está ejecutando, así como el hilo actual y se almacena toda la información. Las invocaciones de método realizan un seguimiento de manera similar. Aunque en la actualidad no está disponible en la interfaz, sería fácil de realizar el seguimiento del número de llamadas a un método específico proporcionado en la entrada (por ejemplo, el seguimiento de los accesos a bases de datos, archivos, conexiones establecidas, etc). Un post-procesamiento se lleva a cabo, finalmente, para mostrar la información recogida para cada uno de los hilos creados.

Para la implementación de los agentes hemos utilizado el interfaz JVMTI. En nuestra aplicación podemos encontrar tres agentes.

El primer agente generado se encarga de obtener los métodos de la clase, que se persigue analizar. Este agente implementa las siguientes funcionalidades:

- Obtención de los métodos:

  Para la obtención de los métodos, hemos implementado uno de los eventos predefinidos por JVMTI. En este caso muchos de los eventos podían sernos útiles, dado que lo único que se requiere es estar en la fase de “vida” de la aplicación (y no. Inicializándose, por ejemplo). Simplemente cuando se produce el evento seleccionado, y comienza el tratamiento del agente, utilizamos la función proporcionada por JVMTI

GetClassMethods, que proporcionándole el nombre de la clase devuelve todos los métodos que implementa. Nosotros los tratamos y los mostramos en la interfaz.

El segundo agente se encarga de realizar la obtención de las variables declaradas y su nombre, dentro del método seleccionado. Este agente implementa las siguientes funcionalidades:

- Obtención de variables locales

  Para ello implementamos el evento "methodEntry", que se produce cada vez que un método es invocado; dentro de este método cribamos para elegir el seleccionado y con la función jvmti "getLocalVariableTable", obtenemos la información de las variables accesibles desde este método.

El tercer agente generado, se encarga de la obtención de toda la información concerniente a la ejecución del proceso, y de modificar las variables (de tipo int) de entrada al método seleccionado. Este agente permite multithreading, y se basa en la recolección de 5 tipos de datos:

- Objetos creados

  Se basa en la creación de una lista que obtiene todos los objetos creados por la JVM, a través de la instrumentación del código, y que nosotros denominaremos "heapTracker". A este heapTracker, conjuntamente con los objetos, añadimos la información del thread que lo ha creado, para posteriormente poder obtener la información de los objetos creados para el método seleccionado, por cada hilo que ha procesado el método.

- Métodos invocados

  Esta información también se almacena en una lista, pero esta vez diferenciada por el thread que ejecuta el método. Cada vez que un método es invocado, se produce el evento "methodEntry", en el cual obtenemos el thread que realiza dicha llamada y el método que lo produjo, si este es el método que nosotros hemos seleccionado para analizar, almacenamos en la lista, el nombre de dicho método o aumentamos la cantidad de llamadas que se han realizado para dicho thread en el caso de que ya exista.

- Bytecodes

  Dentro de la librería JVMTI, existe la posibilidad de realizar un evento cada vez que se ejecuta un nuevo Bytecode (método: SingleStep). Cada vez que se produce dicho método, comprobamos que nos encontramos en el método seleccionado e incrementamos para el hilo correspondiente la cantidad de bytecodes ejecutados.

- Tiempo

Cada vez que se produce un evento de entrada comprobamos, el nombre del método y en caso de que sea el seleccionado, almacenamos el hilo que está ejecutando dicho método. En este momento registramos el tiempo de entrada y de la misma manera el tiempo de salida cuando realizamos el evento de salida de dicho método. Obteniendo el tiempo que tarda el método en ejecutarse.

- Tiempo total

  Esta es la única información independiente de cada thread. Cuando comienza la ejecución del agente se registra el tiempo, y de la misma manera cuando finaliza, obteniendo el tiempo final.

Nuestra implementación, se resume en la obtención de la anterior información, habiendo realizado sucesivas aproximaciones hasta obtener una solución final adaptada a los requisitos necesarios.

Para la realización de nuestro proyecto, elegimos usar el framework JVMTI. Para ilustrar los tipos de desarrollo que se han llevado a cabo, todos estos se centran en la siguiente instrucción:

$$java\ -\mathbf{agentlib}:< agent - lib - name >=< options > "prog\ java"$$

Esta instrucción es ejecutada directamente por la consola de los distintos sistemas operativos que tengan instalada una Máquina Virtual de Java (JVM).

El agente es la parte fundamental del proyecto. Estos agentes se escriben en C o C++, incluyendo las librerías “jvmti.h” y “jni.h”. Esta librería permite la obtención de la información y los eventos interesantes obtenidos por la máquina virtual de java.

El interfaz JVMTI permite la obtención de información a través de dos vías:

- Eventos predefinidos: Estos eventos se implementan en el agente y son invocados cuando la máquina virtual genera el evento relacionado. (Por ejemplo: cuando se entra a un nuevo método)

- Instrumentación del código: JVMTI no incluye algunos eventos que se podría esperar de una interfaz con soporte de perfiles. La interfaz en lugar proporciona apoyo para la instrumentación de bytecodes, la capacidad de alterar los bytecodes ejecutados por la máquina virtual de Java que componen el programa de destino. Por lo general, estas alteraciones son agregar "eventos" para el código de un método, por ejemplo, para añadir, al principio de un método, una llamada a MyProfiler.methodEntered ().

Estos agentes se procesan en paralelo con el proceso java ejecutado por JVM, donde la ejecución del agente se intercala con el proceso java, cuando se producen los eventos definidos en el agente.

Dentro de cada uno de estos eventos, JVMTI ofrece un amplio índice de funciones que permite la obtención de los distintos datos y estructuras necesarias para el estudio del proceso.

Nuestro proyecto contiene varios agentes independientes, cada uno de ellos centrado en la obtención de datos específicos (Por ejemplo: obtener las variables de un método, obtener los métodos de la clase, entre otros).

El agente más complejo es el que hemos creado para la obtención de los todos los objetos creados y los métodos llamados por la aplicación hace uso de la capacidad de JVMTI de introducir Bytecodes.

A continuación se explica más detalladamente y con ejemplos el funcionamiento de los agentes:

```
1   #include "jvmti.h"
2
3   JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *vm, char *options, void *reserved)
4   {
5
6   }
7
8   JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm)
9   {
10
11
12  }
13
```

Ilustración 2 Agente básico

Estas son los métodos básicos para la creación de un agente. Al comienzo del proceso la máquina virtual de java, hace una llamada al método “onload”, donde se establecen los eventos implementados para este agente y los métodos a invocar cuando se producen.

En este caso vamos a visualizar un ejemplo de cómo se implementa el evento Method entry

```
#include "jvmti.h"

void JNICALL cb_method_entry(jvmtiEnv *env,JNIEnv* jni_env,jthread thread,jmethodID method)
{

}

JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *vm, char *options, void *reserved)
{
jvmtiEnv* env;
vm->GetEnv(reinterpret_cast<void**>(&env),JVMTI_VERSION);
jvmtiCapabilities capabilities = { 1 };
jvmtiEventCallbacks callbacks = { 0 };
capabilities.can_generate_method_entry_events = 1;
env->AddCapabilities(&capabilities);

env->SetEventNotificationMode(JVMTI_ENABLE,JVMTI_EVENT_METHOD_ENTRY,NULL);
callbacks.MethodEntry = &cb_method_entry;
env->SetEventCallbacks(&callbacks, sizeof(callbacks));
}

JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm)
{

}
```

Ilustración 3 Agente simple inicialización

Una vez declarado este evento, cuando se produce, se pueden consultar y ejecutar las funciones presentes en el interfaz JVMTI. A continuación se visualiza el ejemplo de cómo obtener la tabla de variables locales cuando se produce el evento de entrada a un método.

```
#include "jvmti.h"

void JNICALL cb_method_entry(jvmtiEnv *env,JNIEnv* jni_env,jthread thread,jmethodID method)
{
jvmtiError errar;
jint entry_count_ptr;
jvmtiLocalVariableEntry* table_ptr;
errar= env->GetLocalVariableTable(method,
            (jint*) &entry_count_ptr,
            (jvmtiLocalVariableEntry**) &table_ptr);
}

JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *vm, char *options, void *reserved)
{
jvmtiEnv* env;
vm->GetEnv(reinterpret_cast<void**>(&env),JVMTI_VERSION);
jvmtiCapabilities capabilities = { 1 };
jvmtiEventCallbacks callbacks = { 0 };
capabilities.can_generate_method_entry_events = 1;
env->AddCapabilities(&capabilities);

env->SetEventNotificationMode(JVMTI_ENABLE,JVMTI_EVENT_METHOD_ENTRY,NULL);
callbacks.MethodEntry = &cb_method_entry;
env->SetEventCallbacks(&callbacks, sizeof(callbacks));
}

JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm)
{
}
```

Ilustración 4 Agente simple inicialización y evento

Para uno de nuestros agentes, incluimos la instrumentación del código necesaria para obtener todos los objetos creados por el proceso. Para JVMTI hay 3 maneras para insertar instrumentación:

1. Static Instrumentation: El archivo de clase es instrumentado antes de que ser cargado en la máquina virtual - por ejemplo, mediante la creación de un directorio duplicado de *.class que han sido modificados para añadir la instrumentación. Este método es extremadamente incómodo y, en general, un agente no puede conocer el origen de los archivos de clases que se pueden cargar.

2. Load-Time Instrumentation: Cuando un archivo de clase se carga por la VM, los bytes sin formato del archivo de clase se envían para la instrumentación con el agente. El evento ClassFileLoadHook, provocado por la carga de clases, proporciona esta funcionalidad. Este mecanismo proporciona acceso eficiente y completo a la instrumentación de una sola vez.

3. Instrumentación dinámica: Una clase que ya se ha cargado (y posiblemente aún en ejecución) se modifica. Esta característica opcional es proporcionada por el evento ClassFileLoadHook, provocada por la llamada a la función RetransformClasses. Las clases pueden ser modificadas varias veces y pueden ser devueltos a su estado original. El mecanismo permite la instrumentación que cambia durante el curso de la ejecución.

En nuestro caso hemos utilizado la instrumentación en tiempo de ejecución. En nuestro agente, por tanto, implementamos el evento ClassFileLoadHook. Este evento se envía cuando la máquina virtual obtiene los datos de la clase, pero antes de que se construya la representación en memoria de esa clase.

Dentro de este evento producimos la instrumentación necesaria para obtener todos los objetos creados a través del método java_crw_demo.

La función java_crw_demo es una función de la biblioteca no nativa de java. Esta función recoge los bytes de datos que componen la clase y devuelve los nuevos bytes de datos modificados para la clase en la memoria. La biblioteca java_crw_demo es una pequeña biblioteca de C para hacer algunas inserciones de bytecode básico (BCI) para las clases. Ninguna disposición del agente puede modificar los datos estáticos globales mientras que el classload está siendo procesado. Durante el evento, los bytes que representan la clase son reemplazados, y en la máquina virtual se cargan los bytes de datos reemplazados. La biblioteca java_crw_demo está limitada no puede agregar métodos, campos, o argumentos a los métodos, ni cambia la interfaz básica o la forma del objeto. La intención de esta función es instrumentar los bytecodes de métodos existentes.

Para llevar a cabo esta instrumentación es necesaria la creación de una clase java que sirve como rastreadora de los eventos, y una inicialización previa de los métodos a instrumentar.

La inicialización se produce en el evento VMStart, el cual es invocado cuando la máquina virtual es inicializada. En este momento JNI está activo, pero la máquina virtual no está todavía totalmente inicializada.

```
359  static void JNICALL cbVMStart(jvmtiEnv *jvmti, JNIEnv *env)
360  {
361      enterCriticalSection(jvmti); {
362          jclass klass; jfieldID field; jint rc;
363          static JNINativeMethod registry[2] = {
364              {STRING(HEAP_TRACKER_native_newobj), "(Ljava/lang/Object;Ljava/lang/Object;)V",(void*)&HEAP_TRACKER_native_newobj},
365              {STRING(HEAP_TRACKER_native_newarr), "(Ljava/lang/Object;Ljava/lang/Object;)V",(void*)&HEAP_TRACKER_native_newarr}
366          };
367
368          klass = (*env)->FindClass(env, STRING(HEAP_TRACKER_class));
369          if ( klass == NULL ) {
370              fatal_error("ERROR: JNI: Cannot find %s with FindClass\n",STRING(HEAP_TRACKER_class));
371          }
372          rc = (*env)->RegisterNatives(env, klass, registry, 2);
373          if ( rc != 0 ) {
374              fatal_error("ERROR: JNI: Cannot register natives for class %s\n",
375                          STRING(HEAP_TRACKER_class));
376          }
377
378          field = (*env)->GetStaticFieldID(env, klass, STRING(HEAP_TRACKER_engaged), "I");
379          if ( field == NULL ) {
380              fatal_error("ERROR: JNI: Cannot get field from %s\n",
381                          STRING(HEAP_TRACKER_class));
382          }
383          (*env)->SetStaticIntField(env, klass, field, 1);
384           gdata->vmStarted = JNI_TRUE;
385
386      } exitCriticalSection(jvmti);
387  }
```

Ilustración 5 Agente instrumentación

Como se visualiza en la imagen el evento VMStart, realiza una llamada a diversas funciones. En primer lugar, la función FindClass se llama para obtener el identificador jclass, de la clase a través de la cual se introducen los nuevos bytecodes. A continuación, los métodos nativos de esa clase se ligan con los del agente con el método Register Natives. El cual registra los métodos nativos con la clase especificada, proporcionando unas estructuras JNINativeMethod que contienen los nombres, los tipos, y los punteros de función de los métodos nativos.

Los otros dos métodos GetStaticFieldID y SetStaticIntField sirven para obtener el valor del atributo estático engaged y establecerlo a 1, respectivamente.

```
1  public class HeapTracker {
2
3      private static int engaged = 0;
4
5      private static native void _newobj(Object thread, Object o);
6      public static void newobj(Object o)
7      {
8          if ( engaged != 0 ) {
9              _newobj(Thread.currentThread(), o);
10         }
11     }
12
13     private static native void _newarr(Object thread, Object a);
14     public static void newarr(Object a)
15     {
16         if ( engaged != 0 ) {
17             _newarr(Thread.currentThread(), a);
18         }
19     }
20
21 }
```

Ilustración 6 Agente Instrumentación 2

Una vez realizados estos pasos iniciales, se ejecuta la función java_crw_demo, que instrumenta el código, por cada clase cargada.

Es por este proceso que cuando se crea un nuevo objeto o un array se llama a los siguientes métodos, obteniendo la información ligada a ellos.

```
333 static void JNICALL
334 HEAP_TRACKER_native_newobj(JNIEnv *env, jclass klass, jthread thread, jobject o)
335 {
336     TraceInfo *tinfo;
337     if ( gdata->vmDead ) {
338         return;
339     }
340     tinfo = findTraceInfo(gdata->jvmti, thread, TRACE_USER);
341     tagObjectWithTraceInfo(gdata->jvmti, o, tinfo);
342 }
343
344
345 static void JNICALL
346 HEAP_TRACKER_native_newarr(JNIEnv *env, jclass klass, jthread thread, jobject a)
347 {
348     TraceInfo *tinfo;
349     if ( gdata->vmDead ) {
350         return;
351     }
352     tinfo = findTraceInfo(gdata->jvmti, thread, TRACE_USER);
353     tagObjectWithTraceInfo(gdata->jvmti, a, tinfo);
354 }
```

Ilustración 7 Agente nuevos eventos

Utilizando, por tanto, las herramientas y métodos proporcionados por JVMTI, se consigue obtener las estructuras de datos y la información necesaria para obtener un análisis de los procesos.

La realización de los eventos necesarios, la obtención de la información necesaria en cada punto, la instrumentación del código, la generación de nuevos eventos y el tratamiento de la información suponen un reto de diseño y comprensión del funcionamiento de la Máquina Virtual de Java. A todo ello habría que incluir el control de las secciones críticas y el acceso y obtención de la información por parte de los distintos hilos que se persiguen analizar y que ejecutan el proceso de manera independiente.

7. Resultados Experimentales

Hemos realizado algunos experimentos preliminares en ejemplos tomados del libro [5], que constituyen patrones de programación representativos de Java multi-hilo. El objetivo de los experimentos es doble: (1) nuestro objetivo es demostrar que la herramienta está trabajando en programas de tamaño mediano y obtiene información significativa para ellos, y (2) queremos comprobar la sobrecarga introducida por el análisis. La Tabla 1 recoge los resultados del análisis de cinco programas dos veces con los mismos datos de entrada. Los resultados de cada ejecución se muestran en una fila diferente. Tamaño de columna muestra el tamaño del código analizado en bytecodes. Columnas #B, #O y # M muestran, respectivamente, el número de instrucciones ejecutadas de bytecodes, el número de objetos creados y el número de métodos invocados. Es interesante observar el comportamiento no determinista de la ejecución en todos los casos: siempre obtenemos un consumo de recursos diferente mediante la ejecución del mismo programa con la misma entrada dos veces. Columna Exec (ms) muestra el tiempo necesario para ejecutar el programa sin el analizador activado y Ana (ms) es el momento de ejecutar el programa con Dynja. La sobrecarga es bastante grande en algunos casos (casi cinco veces más grande para WebServices). Es importante destacar que, desde que nuestros recursos medidos son simbólicos, la sobrecarga introducida por la medida hace influencia nuestros resultados.3

Num.	Size	#B	#O	#M	Exec(ms)	Ana(ms)
1	140	37	4	7	110	498
		70	10	16	120	537
2	158	48	14	11	100	385
		44	15	11	120	541
3	174	28	8	23	160	588
		211	73	46	140	486
4	226	774	382	174	11400	13079
		211	77	46	14000	15156
5	301	90	0	20	300	548
		168	0	26	600	1085

Table 1 Preliminary experiments: (1) TurnPlayers, (2) FibonacciThreads, (3) WebServices, (4) RockScissorsPaper, (5) Producer Consumer

Como hemos visto no se produce la misma salida, ni en el caso de que utilicemos las mismas variables iniciales. La ejecución del programa depende de la política de tratamiento de JVM o de las prioridades que hemos establecido para los hilos. Estos hilos, se ejecutan

concurrentemente, además los valores de las variables pueden estar compartidos por todos, produciéndose una solución distinta dependiendo de la política elegida para cada momento, por la JVM.

8. Conclusiones

Para la consecución correcta de nuestro proyecto, hemos pasado por varias fases de aprendizaje y aplicación.

Primeramente comenzamos con un análisis de los actuales perfiladores en el mercado, así como los distintos framework existentes para realizar una herramienta de "perfilado". Este estudio llevo dos meses, en los cuales analizamos los principales perfiladores que se encuentran en el mercado, llegando a la conclusión que ninguno satisfacía los requisitos que nosotros plateábamos.

Como concusión a estos análisis escogimos el interfaz JVMTI, para obtener y realizar nuestro perfilador, dada su completitud y las facilidades que ofrece a la hora de obtener los resultados esperados.

En los siguientes dos meses fueron de aprendizaje y de continuas pruebas del interfaz JVMTI, para conocer todas sus posibilidades y funcionamiento. Realizamos varios agentes durante este periodo que trataban de obtener las primeras informaciones acerca de las aplicaciones java. Durante este periodo comprendimos, como se utilizaba JVMTI, así como a interconectar todos los elementos que componen el análisis a través de agentes que ofrece JVMTI. Entendiendo las distintas funciones de compilación e interconexión entre los distintos módulos.

Finalizada esta etapa comenzamos el desarrollo de la aplicación, generamos distintos agentes para poder obtener información acerca de todos los datos que necesitábamos recolectar. Realizamos varios agentes y varias aproximaciones la solución óptima, solucionando y evitando errores. Con estos desarrollos obtuvimos dos de los tres perfiladores usado en nuestra aplicación.

Para obtener el tercer perfilador, necesitamos incluir instrumentación del código, que permiten contar y guardar la información de todos los objetos creados, así como su método y el hilo que lo generaba.

Para asumir este reto, primeramente obtuvimos un método, sin tener en cuenta los hilos o las secciones críticas, que obtuviese todos los objetos que se crean en la aplicación a analizar. Este método traceaba todos los objetos creados (heapTracker) en el momento de su creación y los guardaba en una lista que posteriormente tratábamos y mostrábamos.

Cuando implementamos la opción multi-hilo, nos dimos cuenta que todos los objetos de los hilos se solapaban y, por tanto, no podíamos obtener el número de objetos por hilo (requisito fundamental de nuestra aplicación)

Realizamos una primera aproximación intentando generar un "heapTracker" por cada uno de los hilos creados por nosotros. Esta aproximación se realizó de varias maneras. Primeramente copiamos la lista de todos los objetos para cada uno de los hilos, así cada uno tendría una copia exacta de todo el "heapTacker" hasta el momento en que se lanzan los hilos. Esta

implementación se rechazó por ineficiente y demasiado compleja. Se realizó una segunda aproximación tratando de que cada hilo se ejecutase por separado y en el orden que se lanzaron. Consiguiéndose todos los resultados esperados. Pero esta aproximación fallaba, en que no permitía que un hilo modificase la ejecución de otro, a través por ejemplo de una variable global. Finalmente obtuvimos una última versión (implementada en la aplicación), que separaba los objetos no solo por su traza tipo y su traza, sino también por el hilo que lo había creado. Obteniendo al final de la ejecución de la aplicación, todos los objetos creados por cada hilo. Esto nos permite un total paralelismo entre todos los hilos, y la creación del sistema de análisis de nuestro agente. Sobre este agente hemos implementado también la cantidad de bytecodes ejecutados por cada hilo y los métodos a los que llama.

Con ello, hemos presentado un prototipo de implementación de un analizador de dinámico de recursos que obtiene las medidas de costos simbólicos para cada hilo creado a lo largo de la ejecución de un método. Sostenemos que nuestra herramienta proporciona información útil durante el desarrollo del software, ya que permite ser conscientes del consumo de recursos de los distintos hilos. Esto puede ser relevante: para los propósitos de optimización; para entender el comportamiento de los recursos en la presencia de intercalaciones complejas, la programación no determinista, y la memoria compartida, y también para la identificación de posibles cuellos de botella en un sistema. También, Dynja se puede utilizar para evaluar la exactitud del análisis de recursos estático. El análisis de recursos estático tiene por objeto inferir el costo del peor caso de la ejecución de un programa. Como el análisis debe tener en cuenta el costo resultante de todas las intercalaciones posibles, esto puede llevar a una estimación demasiado pesimista. El análisis dinámico de recursos es útil para evaluar cómo estos resultados estáticos son de precisos en comparación con los resultados reales obtenidos por el análisis dinámico de varios datos de entrada que son de alguna manera representativos de una ejecución normal.

En cuanto a las herramientas existentes para el análisis de recursos estáticos, sólo sabemos del sistema COSTABS [1]. Se trata de un análisis de costos para el peor caso para un lenguaje basado en objetos concurrentes, un modelo más simple que la concurrencia multi-hilo en Java. Java. COSTA [2] es un analizador de los recursos estático para Java secuencial, a nuestro entender, un analizador de recursos estáticos para Java multi-hilo todavía no existe. Por el contrario, podemos encontrar una serie de herramientas maduras en tiempo de ejecución de perfiles de Java multi-hilo.

Las más populares son: hprof, un perfilador de código abierto que se incluye con Hotspot de Sun y J9 de IBM; xprof, un perfilador interno de Sun Hotspot; JProfile, un producto comercial de tecnologías EJ; YourKit, un producto comercial de YourKit. Una comparación de la exactitud de los perfiladores se puede encontrar en [6]. La diferencia principal entre nuestro analizador de recursos dinámicos y perfiladores existentes es que recopilemos información simbólica. Estos datos simbólicos tienen la ventaja de ser desacoplados de las variaciones en el entorno de ejecución, tales como la carga sobre los recursos compartidos y velocidades de reloj de hardware. Explicar el rendimiento de las aplicaciones multi-hilo de Java es de hecho bastante complejo, especialmente con la presencia de hardware multinúcleo [7]. Nosotros sostenemos que la información obtenida por un perfilador simbólico, ya que se desacopla del entorno de

ejecución, es absolutamente útil para complementar y posiblemente justificar simbólicamente los resultados obtenidos por el perfilador. Los perfiladores de recursos simbólicos han sido definidos para lenguajes declarativos, tanto en secuencial [3], como en los lenguajes paralelos [4].

9. Referencias

[1] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa,and G. Puebla. COSTABS: A Cost and Termination Analyzer for ABS. In Procs. of PEPM'12, pages 151-154. ACM Press, 2012.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, 16th European Symposium on Programming, ESOP'07, volume 4421 of Lecture Notes in Computer Science, pages 157-172. Springer, March 2007.

[3] Elvira Albert and Germán Vidal. Symbolic profiling for multi-paradigm declarative languages. In LOPSTR, volume 2372 of Lecture Notes in Computer Science, pages 148-167. Springer, 2001.

[4] David J. King, Jon G. Hall, and Philip W. Trinder. A strategic profiler for glasgow parallel haskell. In IFL, volume 1595 of Lecture Notes in Computer Science, pages 88-102. Springer, 1999.

[5] D. Lea. Concurrent programming in Java – design principles and patterns. Java series. Addison-Wesley-Longman, 1997.

[6] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In PLDI, pages 187-197. ACM, 2010.

[7] J. B. Sartor and L. Eeckhout. Exploring multi-threaded Java application performance on multicore hardware. In OOPSLA, pages 281-296. ACM, 2012.

[8] Herb Sutter and James R. Larus. Software and the concurrency revolution. ACM Queue, 3(7):54-62, 2005.

[9] Descripción JVMTI [http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html]

10. Apéndice

A Dynamic Resource Analyzer for Multi-Threaded Java

Elvira Albert
U. Complutense of Madrid
elvira@fdi.ucm.es

Iván Troyano
U. Complutense of Madrid
ivantroyano@ucm.es

Oscar Troyano
U. Complutense of Madrid
oscartroyano@ucm.es

ABSTRACT

We present the concepts, usage and prototypical implementation of DYNJA, a dynamic resource analyzer for multi-threaded Java. The system receives as input a Java application, initial values for its input parameters, and the *cost metrics* to be measured among the three metrics currently available: number of executed bytecode instructions, number (and type) of objects created, and number (and name) of methods invoked. DYNJA yields as output the resources consumed by each thread according to the selected metric(s). Our dynamic resource analyzer has been implemented using the Java Virtual Machine Tool Interface (JVMTI), a native programming interface which allows inspecting the state and controlling the execution of applications running in a JVM.

Categories and Subject Descriptors

F3.2 [**Logics and Meaning of Programs**]: Program Analysis; F2.9 [**Analysis of Algorithms and Problem Complexity**]: General; D.1.3 [**Programming Techniques**]: [Concurrent Programming] *Distributed programming, Parallel programming*

General Terms

Languages, Theory, Verification, Reliability

Keywords

Dynamic Analysis, Resource Guarantees, Profiling, Multi-Threaded Java, JVMTI

1. INTRODUCTION

It is widely recognized that multi-threaded programs inherently have a complex behavior (see e.g. [8]). One of the main reasons for such complex behaviour is related to non-deterministic thread scheduling and preemption. Java threads are generally preemptive between priorities. A higher priority thread takes precedence over a lower priority thread. If a higher priority thread goes to sleep or blocks, then a lower priority thread can run. However, as soon as the higher priority thread wakes up or unblocks, it will interrupt the lower priority thread and run until it finishes, until it blocks again, or until it is preempted by a higher priority thread. The Java Language Specification allows VMs to occasionally run a lower priority thread instead of a higher priority one, but in practice this is unusual. Finally, nothing in the Java Language Specification specifies what happens with equal priority threads. All in all, at a conceptual level, one can only assume that scheduling is non-deterministic.

The main difficulty with non-deterministic scheduling and preemption is that, at any point of a thread's execution, the shared-memory may be modified by an interleaved thread. As we will see later, such modifications will affect the behavior of the program and, in particular, can change its cost or *resource consumption*. To build more secure, predictable and optimized multi-threaded systems, we need tools that help understanding and verifying multi-threaded programs, in both their functional and also non-functional aspects. In this paper, we focus on the resource consumption of programs, one of the most important non-functional properties. We consider three *cost metrics* which allow us to measure: the number of executed bytecode instructions, number (and types) of objects created, and number (and names) of invocations to methods.

There are two main approaches to estimate the resource consumption: static and dynamic analyses. Static analysis aims at estimating the resource consumption without executing the program, by only inspecting the program code. The results of static analysis must be *sound* for any execution of the program. Thus, this approach must consider the resource consumption of all feasible paths of execution (and interleavings) and avoid leaving any unchecked behavior. The analysis will return the worst-case cost of the information obtained from all paths. Due to the vast number of feasible interleavings a large multi-threaded system may exhibit, it can lose precision quickly (lead to too pessimistic resource estimation). Dynamic analysis (a.k.a. profiling) gathers information from running the system. It can collect precise and fine-grained behavioral data from a running multi-threaded system, which can be coupled with a post-processing to summarize and reason about the observed results. In general, the collected data is accurate of system execution as long as the overhead of the measurement has not influenced the results. Profiling is limited, however, to the inspection of behavior that can be made by running the system on a selection of input. This limitation means that

PPPJ'13, Stuttgart, Germany, September 11-13, 2013.

profiling is useful in circumstances where a sampling of behavior is sufficient. This can be the case for estimations on the *resource consumption* on statistically frequent paths of execution, but it is not well suited to ensure correct behavior in a system when only one execution in a million can lead to system failure.

Approaches to profiling can be classified as either active or passive. Active profiling requires that the application being measured explicitly generates information about its execution. Passive profiling relies on explicit inspection of control flow and execution state through an external entity. Passive profiling does not require any modification of the measured system, but is harder to implement. DYNJA is a passive, dynamic, resource analyzer for multi-threaded Java which has, among others, interesting applications to (1) understand the resource consumption of each thread in the presence of complex thread interleavings that may affect the cost, (2) to assess the accuracy of static resource analysis which infers worst-case resource consumption, (3) to explain *symbolically* the reason of hot-spots in the execution (i.e., which thread is using more resources and of what type). The DYNJA system can be downloaded from `http://costa.ls.fi.upm.es/dynja` where the examples used in the paper can be found as well.

The remainder of the paper is organized as follows. Section 2 describes the interface of our tool, and shows on a working example the kind of information that DYNJA can yield in its output. Section 3 provides some implementation details. Section 4 presents preliminary experiments of our prototype on examples taken from [5]. Finally, Section 5 concludes and reviews some related tools.

2. DESCRIPTION OF DYNJA

The DYNJA system can be used through its Java interface which is depicted in Figure 2. The user first selects the application to be analyzed. After pressing the button `compile`, DYNJA compiles it using the JVM installation on the machine, and identifies the methods defined in the application. The list of methods that are available are displayed in the list "Method to profile" that appears next.

EXAMPLE 1. *As an example, consider the following classes* `MyThread` *and* `ResThreads`*. Objects of type* `MyThread` *have a field* `ini` *and a field* `t` *which are initialized on construction. As* `MyThread` *extends class* `Thread`*, objects of type* `MyThread` *can be run via method* `run` *which invokes method* `task` *on the field object* `t`*. Concurrent interleavings may happen when executing* `task` *on several threads. The interesting issues in method* `task` *are (1) the use of shared data, namely* `bound` *and* `z` *are class fields that will take different values depending on the scheduling policy and on how threads interleave and this will affect the resource consumption, (2) a division by* `0` *in line 30 can occur depending on the scheduling and on how threads interleave, affecting again the resource consumption. Method* `main` *receives three integer data via its input parameter* `argv`*, creates a single object* `tester` *that is used later by the two threads created at lines 38 and 42. These threads are started after creation and the* `run` *method starts to execute on each of them. As the threads share object* `tester`*, the scheduling strategy and threads interleavings might change the values of the fields of* `tester` *in different ways. This affects the resource consumption as we illustrate below.*

We can see in the Java interface of DYNJA that, if the

```
1 public class MyThread extends Thread{
2   int ini;
3   ResThreads t;
4   public MyThread(String name, ResThreads t, int ini)
        {
5     super(name);
6     this.t = t;
7     this.ini=ini;
8   }
9   public void run() {
10    t.task(ini);
11  }
12 }
13 public class ResThreads{
14 int z;
15 int bound;
16 public ResThreads(int z,int bound){
17    this.z=z;
18    this.bound=bound;
19 }
20 public void task(int ini){
21  while(ini<bound)
22    {
23    if (ini ==1) for(int j =0; j<10000;j++);
24      Integer obj = new Integer(ini);
25      obj.intValue();
26      ini++;
27     }
28    bound--;
29    int r= z-ini;
30    int exc = 2/r;
31    // not executed if exception
32 }
33 public static void main(String argv[]){
34    int ini = Integer.parseInt(argv[0]);
35    int bound = Integer.parseInt(argv[1]);
36    int z = Integer.parseInt(argv[2]);
37    ResThreads tester = new ResThreads(z,bound);
38    Thread t1 = new MyThread("thread1",tester,ini);
39    t1.setPriority(Thread.MIN_PRIORITY);
40    t1.start();
41    for(int j =0; j<4;j++);
42    Thread t2 = new MyThread("thread2",tester,ini+1);
43    t2.setPriority(Thread.MAX_PRIORITY);
44    t2.start();
45 }
46
```

Figure 1: Running Multi-Threaded Java Example

method to be analyzed is `main`, input values are provided in a specific TextField by separating the values by blank spaces. In the example, we have provided as initial values 1 3 2. In addition, we might optionally specify fixed input values for parameters and variables of some other method in the application (which is different from `main`). By pressing on the button "Show variables", we visualize all variables defined in the selected method, including the input parameters. For instance, we can force that `task` be always executed with some initial value X as input parameter. The value X is provided using the syntax "ini=X" in the corresponding TextField. In Figure 2, we have only provided initial values for `main`, thus analysis starts from the execution of `main` and `task` is executed with the values that the parameters actually have in the calls to `task` reached from `main`. Thus, the selection of `task` has been superfluous in this case.

The next option refers to the *cost metrics*, i.e., the resource that we want to measure. We can select the follow-

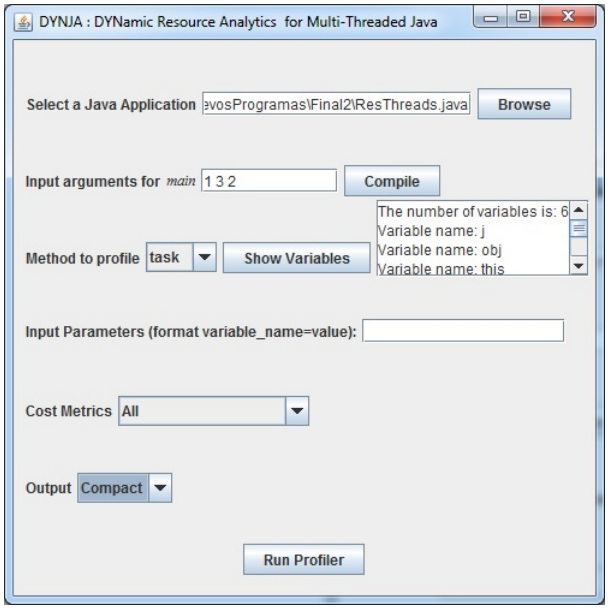


Figure 2: Basic Interface of DYNJA

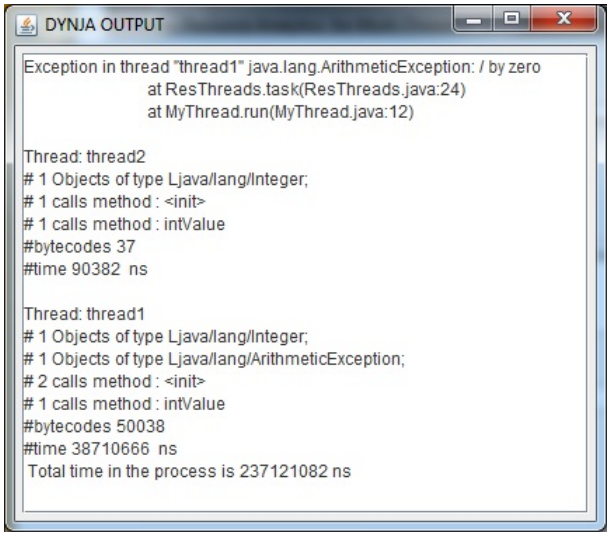


Figure 3: Compact Output of DYNJA

ing *symbolic* cost metrics which are measured on each of the threads created (our metrics are symbolic in the sense that they are independent of the execution environment):

- *bytecode*: the number of executed bytecode instructions on each thread;
- *objects*: the number of created objects of each type;
- *calls*: the number of method invocations to the different methods;
- *all*: all above metrics.

The last input parameter of DYNJA allows selecting if the output of the runtime analysis is showed in *compact* or *verbose* mode. The compact mode for our example can be seen in Figure 3, where we see the result of the cost metrics on each thread, as well as the thread execution time. Let us explain the analysis results. First, observe that we have added a loop at line 41 to let the low-priority thread `t1` start to execute so that we can see that the high-priority thread `t2` later interrupts its execution. Also, the loop in line 23 is executed only in `t1` (hence this thread executes many more bytecode instructions than `t2`). The motivation of adding this loop is that it allows us to capture the fact that the loop bound `bound` can be modified by the interleaved thread in the middle of the loop execution. In the output, we indeed observe that, due to such thread interleaving, `t1` throws an exception because `t2` decreases the value of `bound` to 2 and, thus `ini` increases only up to 3 in `t1`, and then `r` takes value 0. This causes a division by 0 in `t1` and thus the subsequent code (which is commented out in line 31) will not add any further resource consumption.

Note also that the number of iterations of the while loop at line 21 varies depending on thread interleaving, and this affects the number of objects created at line 24 and the number of calls at line 25. In particular, `t1` creates only 1 object and makes 1 method call to `intValue` (while without interleaving it should have consumed 2 of each type) since `bound` was decreased by the interleaved `t2`. The verbose mode additionally dumps heap trace information that is gathered by the JVMTI interface. An interesting aspect is that consecutive executions of DYNJA with the same input data can lead to different resource consumption values. This happens for instance if we remove the two lines 39 and 43 with the `setPriority` invocations and also the delay added at line 41. As the threads have the same priority, some executions start by `t1` and other executions by `t2` non-deterministically.

3. IMPLEMENTATION

We have implemented the DYNJA tool using the Java Virtual Machine Tool Interface (JVMTI) which provides access and control on the JVM under which the program is run. This allows that the analyzer is decoupled from the JVM that executes and can be used on different JVMs and platforms. JVMTI provides a programming interface that allows creating software agents that monitor and control Java applications. The purpose of JVMTI is to enable the creation of debugging and profiling tools for Java applications.

As regards implementation, JVMTI is a native interface of the JVM. A library, written in C or C++, is loaded during the initialization of the JVM. The library has access to the JVM state by calling JVMTI and JNI (Java Native Interface) functions and can register to receive JVMTI events using event handler functions that are called by the JVM when such an event occurs. In spite of the predefined events that JVMTI already provides, the development of DYNJA has required the instrumentation of the bytecode instructions associated to object and array creation in order to be able to track for each thread the objects and arrays that it creates. Then, the main part of our implementation is an *agent* which uses some functionality exposed by the interface to register for notification of events associated to our cost metrics as they occur in the application. Essentially, when an event occurs (e.g., an object is created), it adds the information on the thread that has created it. When a bytecode instruction is executed, it checks the method that is executing as well as the current thread and stores all such information. Method invocations are tracked in a similar

way. Although it is currently not available in the interface, it would be straightforward to track the number of calls to a specific method provided in the input (e.g., tracking accesses to databases, files, connections established, etc.). A post-processing is finally performed to display the gathered information for each of the created threads.

4. PRELIMINARY EXPERIMENTS

We have performed some preliminary experiments on examples taken from the book [5], which constitute representative programming patterns for multi-threaded Java. The goal of the experiments is twofold: (1) we aim at showing that the tool is working on medium-sized programs and obtains meaningful information for them, and (2) we want to check the overhead introduced by the analysis. Table 1 collects the results of analyzing five programs twice with the same input data. The results for each execution are showed in a different row. Column Size shows the size of the analyzed code in bytecodes. Columns #B, #O and #M show, respectively, the number of executed bytecode instructions, the number of objects created and the number of methods invoked. It is interesting to observe the non-deterministic behaviour of the execution in all cases: we always obtain a different resource consumption by executing the same program with the same input twice. Column Exec(ms) shows the time taken to run the program without the analyzer activated, and Ana(ms) is the time to run the program with DYNJA. The overhead is rather large in some cases (almost five times larger for **WebServices**). Importantly, since our measured resources are symbolic, the overhead introduced by the measurement does influence our results.

Num.	Size	#B	#O	#M	Exec(ms)	Ana(ms)
1	140	37	4	7	110	498
		70	10	16	120	537
2	158	48	14	11	100	385
		44	15	11	120	541
3	174	28	8	23	160	588
		211	73	46	140	486
4	226	774	382	174	11400	13079
		211	77	46	14000	15156
5	301	90	0	20	300	548
		168	0	26	600	1085

Table 1: Preliminary experiments: (1) TurnPlayers, (2) FibonacciThreads, (3) WebServices, (4) RockScissorsPaper, (5) Producer-Consumer

5. CONCLUSIONS AND RELATED TOOLS

We have presented a prototype implementation of a dynamic resource analyzer that obtains *symbolic* cost measures for each thread created along a method's execution. We argue our tool provides useful information during software development, as it allows being aware of the resource consumption of the different threads. This can be relevant: for optimization purposes; for understanding the resource behaviour in the presence of complex interleavings, non-deterministic scheduling, and shared memory; and also for identifying potential bottlenecks in a system. Also, DYNJA can be used to assess the accuracy of static resource analysis. Static resource analysis aims at inferring the worst-case cost of executing a program. As analysis needs to consider the cost resulting from all possible interleavings, this can lead to a too pessimistic estimation. Dynamic resource analysis is useful to assess how precise static results are when compared to actual results obtained by dynamic analysis on several input data that are somehow representative of a normal execution.

As regards existing tools for static resource analysis, we only know of the COSTABS system [1]. This is a worst-case cost analysis for a language based on concurrent objects, a concurrency model simpler than multi-threaded Java concurrency. COSTA [2] is a static resource analysis for sequential Java but, to our knowledge, a static resource analyzer for multi-threaded Java does not exist yet. We can find a series of mature tools for runtime profiling multi-threaded Java. The most popular ones are: *hprof*, an open-source profiler that ships with Sun's Hotspot and IBM's J9; *xprof*, an internal profiler in Sun's Hotspot; *jprofile*, a commercial product from EJ technologies; *yourkit*, a commercial product from YourKit. A comparison of the accuracy of the profilers can be found in [6]. The main difference between our dynamic resource analyzer and existing profilers is that we collect symbolic information. Symbolic data has the advantage of being decoupled of variations in the execution environment, such as load on shared resources and hardware clock speeds. Explaining the performance of Java multi-threaded applications is indeed rather complex, specially in the presence of multicore hardware [7]. We argue that the information obtained by symbolic profiling, as it is decoupled from the execution environment, is useful to complement and possibly justify symbolically the results gathered by profiling. Symbolic resource profilers have been defined for declarative languages, both for sequential [3] and parallel languages [4].

6. REFERENCES

[1] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: A Cost and Termination Analyzer for ABS. In *Procs. of PEPM'12*, pages 151–154. ACM Press, 2012.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[3] Elvira Albert and Germán Vidal. Symbolic profiling for multi-paradigm declarative languages. In *LOPSTR*, volume 2372 of *Lecture Notes in Computer Science*, pages 148–167. Springer, 2001.

[4] David J. King, Jon G. Hall, and Philip W. Trinder. A strategic profiler for glasgow parallel haskell. In *IFL*, volume 1595 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 1999.

[5] D. Lea. *Concurrent programming in Java - design principles and patterns*. Java series. Addison-Wesley-Longman, 1997.

[6] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In *PLDI*, pages 187–197. ACM, 2010.

[7] J. B. Sartor and L. Eeckhout. Exploring multi-threaded Java application performance on multicore hardware. In *OOPSLA*, pages 281–296. ACM, 2012.

[8] Herb Sutter and James R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.